Winter 1998

A Hierarchical Filtering-Based Monitoring Architecture for Large-scale Distributed Systems

Ehab Salem Al-Shaer
Old Dominion University

A HIERARCHICAL FILTERING-BASED MONITORING ARCHITECTURE FOR LARGE-SCALE DISTRIBUTED SYSTEMS

by

Ehab Salem Al-Shaer
B.Sc., May 1990, King Fahad University of Petroleum and Minerals, Saudi Arabia
M.Sc., December 1993, Northeastern University, Boston, Massachusetts

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 1998

Approved by:

Hussein Abdel-Wahab (Director)

Kurt Maly (Director)

Shunichi Toida (Member)

J. Christian Wild, Jr. (Member)

Martin Meyer (Member)

ABSTRACT

HIERARCHICAL FILTERING-BASED MONITORING ARCHITECTURE
FOR LARGE-SCALE DISTRIBUTED SYSTEMS

Ehab Salem Al-Shaer
Old Dominion University, 1998
Co-Directors: Dr. Hussein Abdel-Wahab
Dr. Kurt Maly

On-line monitoring is essential for observing and improving the reliability and performance of large-scale distributed (LSD) systems. In an LSD environment, large numbers of events are generated by system components during their execution and interaction with external objects (e.g. users or processes). These events must be monitored to accurately determine the run-time behavior of an LSD system and to obtain status information that is required for debugging and steering applications. However, the manner in which events are generated in an LSD system is complex and represents a number of challenges for an on-line monitoring system. Correlated events are generated concurrently and can occur at multiple locations distributed throughout the environment. This makes monitoring an intricate task and complicates the management decision process. Furthermore, the large number of entities and the geographical distribution inherent with LSD systems increases the difficulty of addressing traditional issues, such as performance bottlenecks, scalability, and application perturbation.

This dissertation proposes a *scalable, high-performance, dynamic, flexible* and *non-intrusive* monitoring architecture for LSD systems. The resulting architecture detects and classifies interesting primitive and composite events and performs either a corrective or steering action. When appropriate, information is disseminated to management applications, such as reactive control and debugging tools.

The monitoring architecture employs a novel hierarchical event filtering approach that distributes the monitoring load and limits event propagation. This significantly improves *scalability* and *performance* while minimizing the monitoring *intrusiveness*. The architecture provides *dynamic* monitoring capabilities through: subscription policies that enable applications developers to add, delete and modify monitoring demands on-the-

fly, an adaptable configuration that accommodates environmental changes, and a programmable environment that facilitates development of self-directed monitoring tasks. Increased *flexibility* is achieved through a declarative and comprehensive monitoring language, a simple code instrumentation process, and automated monitoring administration. These elements substantially relieve the burden imposed by using on-line distributed monitoring systems. In addition, the monitoring system provides techniques to manage the trade-offs between various monitoring objectives.

The proposed solution offers improvements over related works by presenting a comprehensive architecture that considers the requirements and implied objectives for monitoring large-scale distributed systems. This architecture is referred to as the *HiFi* monitoring system.

To demonstrate effectiveness at debugging and steering LSD systems, the HiFi monitoring system has been implemented at the Old Dominion University for monitoring the Interactive Remote Instruction (IRI) system. The results from this case study validate that the HiFi system achieves the objectives outlined in this thesis.

To

Mother, Father, Ruba and Abrar

ACKNOWLEDGMENTS

I can not possibly thank enough my advisors, Professor Hussein Abdel-Wahab and Professor Kurt Maly, for their guidance and support through the tortuous path of the Ph.D. I am deeply indebted to them for a lot of long discussions and feedback throughout the evolution of this work. Many thanks must go to my committee members, Professor Shunichi Toida, Professor J. Christian Wild, Jr and Professor Martin Meyer for constructive criticism on the dissertation. I would also thank all faculty and colleagues in IRI Lab at the Computer Science Department of Old Dominion University for their support and encouragement.

Last, but not least, I would never have finished this work without the moral support, patient, and encouragement of my parents (Salem and Fayzah), my wife (Ruba) and my daughter (Abrar) to whom I dedicated this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

"*We reject kings, presidents and voting; we believe in rough consensus and* **running code.**"

– Dr. David Clark (July 18 1992)

The networked and distributed systems decade is upon us. Computers are no longer used as stand-alone devices. Over the past two decades, there have been tremendous research and development efforts in the areas of high-speed networking, protocols, group communication, video teleconferencing applications, and collaborative distributed applications. More and more applications continue to be deployed over the MBone and Internet, which has attracted our imagination and earned our admiration. Respectful of the huge collective effort to deploy applications and services, we speculate and wonder about the future of such scattered applications without "real" management support. In this thesis, we address the problems posed by monitoring such large-scale distributed services. A distinguishing trait of this thesis work is the particular emphasis that we placed on producing a real monitoring system which we believe will serve as a valuable vehicle for managing large-scale distributed applications.

1.1 Motivation

The demands of large-scale distributed (LSD) systems are increasing. Two influential factors encourage employing LSD applications in many domains: advances in Internet and Intranet technologies, and the economical and performance benefits of distributed applications. Examples of LSD systems include large-scale collaborative distance learning, video

The journal model for this dissertation is the *IEEE/ACM Transactions on Networking.*

teleconferencing, reliable multicasting, distributed transaction systems, distributed interactive simulation, interactive multi-party games, and Internet distributed services, such as Internet service providers (ISP) and digital libraries. LSD systems involve a large number of users or application entities that are geographically dispersed over interconnected LANs (i.e., Intranets) or over WANs (i.e., Internet). Although these applications enable interaction and resource sharing without regard to geographical distances, they inherit distributed processing problems. Particularly, they support a large number of interactions and span any number of networks comprised of widely varying state and configuration. Writing a distributed program that *sometimes* behaves "correctly" differs completely from writing one that performs "well". The former executes without failure and provides the correct output but in a manner that is unpredictable based on the application environment used. The latter not only behaves well but also knows how to adapt itself to sustain predictable behavior given wide ranges of environmental and operational parameters. Developing large-scale distributed applications that perform "well" is especially difficult. *Reliability* and *performance* of applications become critical issues given the distributed nature and large number of participants, or application entities, in an LSD system. Wide geographical distribution and high amounts of interaction may increase the possibility of failures or errors while also increasing the likelihood of performance bottlenecks.

Traditionally, the execution of distributed programs is manually monitored by inserting hardwired statements into application code. These statements generate run-time traces and then collect and analyze trace data to identify answers for unresolved problems. While this tedious and error-prone process is feasible, it provides opportunities only for *ad hoc* fixes, which ultimately results in poorly engineered systems that exhibit unpredictable performance and/or inconsistent behavior.

On-line monitoring is an essential means for improving system reliability and performance. This is due to its effectiveness in observing the run-time behavior of distributed applications and for providing feedback information, which is required to accurately identify and resolve problems, to management units. Management units can be human system managers or automated software components that require feedback for initiating corrective actions. Corrective actions can be performed either at *run-time*, as is the case with applications steering and fault recovery (i.e., reactive control), or at *development-time*, when program bugs are being fixed or designs are being enhanced (i.e., debugging).

A large-scale distributed system must be monitored throughout its execution so that program behavior is revealed based on state changes and failures. Monitoring information, as represented by *events*, is sent from an executing program or generated during interaction with external objects, such as users or other applications to the management units. These events represent the run-time behavior of LSD systems. In non-distributed systems, developers express these events via "print" statements or by using any generic debugger tool (e.g. gdb) for monitoring and inspecting application behavior.

Monitoring distributed systems is very complex because events are distributed throughout the application environment and happen concurrently. An event representing an application failure, or other condition, may be a correlation of events occurring in various locations of the distributed environment. For example, failures in the communications operation may require observing events from multiple senders and receivers. As such, several events may be sent which correspond with the failure and, in aggregate, describe the observed system behavior. Similarly, knowledge of performance bottlenecks is also distributed in the application environment. For instance, in reliable multicasting, discovery of slow members requires current performance information (feedback) from most, if not all, members in the group. This data is captured via event generation, which happens in a concurrent fashion throughout the environment.

The large volume of event producers and management applications in an LSD system may overload the monitoring system. Additionally, the propagation of events over the entire network may cause perturbations in the applications' execution. By the very nature of an LSD system, monitoring these systems is inherently distributed. This often imposes difficulty in reconfiguring and administering remote monitoring entities and often makes the management process incomprehensible. These attributes highlight the complexity and difficulty of developing effective monitoring systems for large-scale distributed environments.

This work is motivated primarily by (1) a requirement for an efficient architecture for monitoring large-scale distributed systems, which will provide reliability and performance improvements for LSD applications, and (2) the lack of an existing monitoring system that meets design goals and satisfies the requirements of LSD systems.

1.2 Problem Description

A distributed system is defined as a collection of autonomous processors and data stores that interact cooperatively to achieve an overall goal [81]. However, this definition is general and includes shared-memory and message-passing distributed systems. In the context of our research, the definition of distributed systems is limited to those that interact via *message passing* only [88]. In this view, distributed systems are systems running on cluster of workstations interconnected via LAN, WAN or Intranets of LAN and WAN. A "large-scale" distributed system, in the context of this dissertation, is a distributed system that includes any of the following attributes: (1) large geographical distribution, (2) large numbers of application entities (processes) or users, or (3) multiple services of differing priority classes such as distributed multimedia applications. Many large-scale distributed systems, such as interactive distance learning and distributed interactive simulation (DIS), exhibit some or all of these attributes.

The primary goal is to design an efficient architecture for monitoring large-scale distributed systems. It is important at this stage to understand the characteristics and requirements of large-scale distributed systems that impact an effective design. These characteristics and requirements are outlined below.

- **Correlated events may be concurrent and distributed:** Monitoring the behavior of distributed systems may require analyzing and correlating events from different sources and generated at different times. It is mandatory for distributed system monitors to be capable of detecting both primitive and composite events. Primitive events are comprised of a single notification while composite events represent a correlation of two or more notification events. However, unlike many of the proposed monitoring systems [27, 29, 60, 69, 78, 93], an assumption of a centralized server or global state at which monitoring information (events) are collected or analyzed is not presumed. Thus, this requires developing an efficient mechanism to abstract and model types of events that can be generated in LSD systems and subsequently direct the monitoring system to detect and classify both event types is needed.

- **Large number of event producers:** LSD systems may contain a large number of event producers that continuously send notifications expressing their behavior. In

addition, the event producers may generate a high volume of event notifications that could swamp the monitoring system. This implies that the monitoring system must be *scalable* and *efficient* to handle a large number of producers and A high-volume of generated events.

- **Different event consumers and different views:** Managing LSD systems is a complex task and may involve a significant number of event consumers, such as managers or management units. However, each event consumer may be dedicated for different missions and monitoring views. This implies that the system must be (1) *scalable* to handle the number of consumers requests and (2) highly *re-configurable* to handle dynamic consumer requests (run-time add, delete and modify) and to disseminate the monitoring information with minimal performance overhead.

- **Wide geographical distribution:** No assumption is to be made about the physical or geographical distribution of a monitored distributed system. Target systems are usually distributed over large distances, such as is the case with distance learning applications. Therefore, employing mechanisms that limit event propagation is substantially important in large networks. It is also essential in a distributed environment to provide powerful and automatic remote configuration tools that facilitate the administration and manageability of the monitoring system. This is different from the previous work that supports monitoring systems for parallel programs with shared memory [32], or for clusters of workstations connected in the same LAN [58, 69].

- **Monitoring application behavior:** The monitoring system architecture is highly influenced by the target management application supported by the system. For example, if the desired monitoring service is to solely measure the CPU and memory utilization of a distributed system, then monitoring programs such as `top`, `gprof` or `Quantify` [73] is sufficient. The only requirement is to combine captured information from the various systems into one report. However, *external monitors* are inadequate if the goal is to monitor the behavior of a distributed system for debugging or reactive control services. In this case, the internal state of the running systems needs to be investigated based on the requests of the monitoring application. We refer to this as *internal state monitors.*

- **Limited resources:** The monitoring system may compete for system resources (i.e., processor cycles or memory for computation or network resources) with monitored objects. Distributed systems always have limited system resources and the system architecture must provide a solution for limiting the intrusiveness in the application environment.

- **Different service priority:** Today, many distributed systems comprise services and components that have different time-constraints. These applications include distributed group-aware multimedia applications that support multiple, concurrent streams of audio, video and data, each of which has its own requirement for quality of service.

After defining target applications, and describing the characteristics and requirements of such applications, the problem to be addressed by this dissertation can be stated as: *"Designing and developing an efficient architecture that considers the requirements and implied objectives for monitoring large-scale distributed systems"*. In other words, "What is the *optimal* monitoring architecture, design and implementation for large-scale distributed systems? And how can it be deployed for useful application in managing distributed systems?". This dissertation attempts to address these questions by making a sound argument that our proposed architecture provides steps towards an efficient solution to the stated problem. The next section defines work objectives based on these requirements specifications.

1.3 Objectives

The high-level objective of this research is to *design and develop a dynamic monitoring architecture for large-scale distributed systems by efficiently classifying primitive and composite events generated by these applications during execution.* The design must be *dynamic* since consumers can spontaneously change subscriptions at run-time, and also must be *efficient* as the monitoring system must utilize a scalable, high-performance, flexible and non-intrusive architecture. The monitoring architecture should support debugging applications such as dynamic and customized traces. Support should also be provided for reactive control services, such as adaptable application steering that effectively improves

the reliability and performance of LSD systems. The design objectives of the presented monitoring architecture are:

Supporting efficient monitoring architecture. The efficiency of the monitoring architecture is measured by its capability to be "high-performance", to handle large volumes of events, and to be "scalable" to accommodate large numbers of producers (i.e., application entities) and consumers, which is typical in the LSD system environment.

Detecting primitive and composite events. The monitoring system must be capable of classifying simple primitive (e.g. local) events as well as complex composite (e.g. global) events specified in AND/OR regular expressions. Consumers may need to know if certain patterns of events occur under certain conditions. For this reason, the monitoring system should not only classify primitive events, but also track the global event history of the system to detect specified combinations of events.

Supporting dynamic monitoring subscription. The consumers' subscriptions may be changed dynamically at run-time. In other words, consumers may add, delete and modify their monitoring subscriptions dynamically at run-time. Therefore, the monitoring architecture must be *dynamically* reconfigurable to ensure spontaneous response to users' subscriptions. Since different consumers may have different concerns and subscribe to different monitoring information, an efficient dissemination mechanism is needed in order to reduce the overhead of distributing the monitoring information.

Minimizing intervention. The monitored applications must be executed with special instructions inserted in the code in order for monitoring services to occur. This is called the *instrumentation* process. The monitoring architecture must provide (1) a simple technique to instrument the monitored object with minimum involvement from consumers (e.g., developers), (2) a flexible and dynamic event reporting process, and (3) a highly manageable monitoring infrastructure that facilitates starting, controlling and administrating monitoring system components.

Minimizing the monitoring intrusiveness. The proposed monitoring architecture

must efficiently monitor LSD system so that system resources (computational and networked) are not overwhelmed. Consideration has to be given in the architecture to minimize perturbation and intrusiveness in the monitoring system. As has been stated: "The impact of adding network management to managed nodes must be minimal, reflecting a lowest common denominator" [75].

Supporting reactive control services. A reactive control system enables consumers to define specific actions to be triggered when certain events (primitive or composite) are detected. We call this process the *action service.* The action service is necessary for monitoring applications, such as application steering. The proposed architecture must support a mechanism to provide this service.

Supporting priority-based monitoring. In LSD systems environment, events typically have different levels of importance to users. Consumers wish to monitor (process and forward) events in accordance with their priorities.

1.4 Architecture Overview: User's Perspective

This dissertation presents a distributed hierarchical monitoring architecture that addresses the objectives described in the previous section. This new architecture attempts to fully utilize the hierarchical filtering monitoring approach so that monitoring load is distributed throughout the monitoring environment.

To provide a comprehensive architecture, the monitoring system provides four fundamental services: the *instrumentation service* which facilitates program preparation and environment configuration, the *monitoring subscription service* which enables users to define their demands in a flexible manner, the *event filtering service* which uses collaborative agents to filter events based on a hierarchical organization and in compliance with appropriate management protocols, and the *action service* which supports management applications such as fault recovery and reactive control.

Event consumers (e.g., users) start the monitoring process by instrumenting their programs. When programs are executed, the monitoring agent hierarchy is established and monitoring operations are initiated. This hierarchy of agents serves as an intermediate

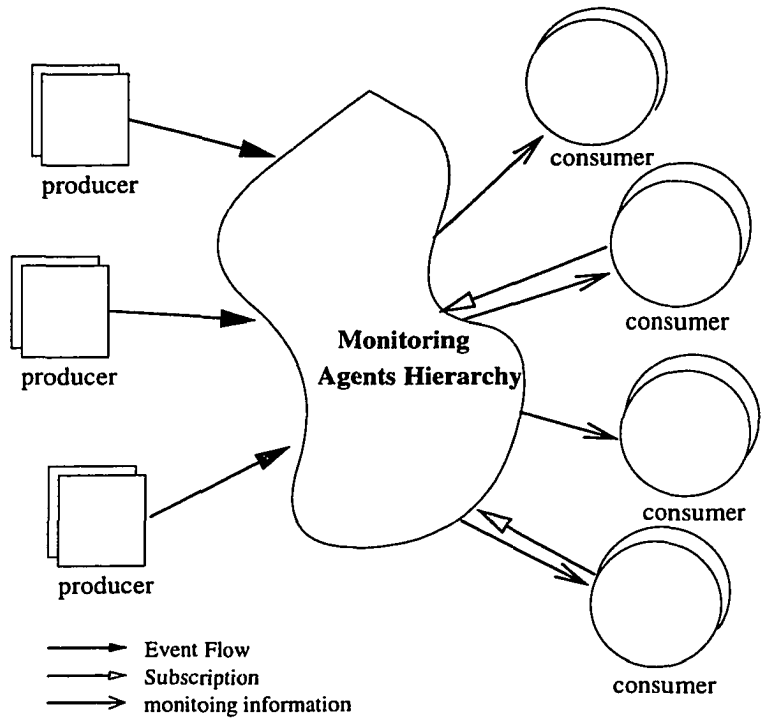


Fig. 1.1. Basic Monitoring Model.

broker between user requests and application events (see Figure 1.1). Once monitoring operations are established, users receive confirmation and may begin interacting with the monitoring system (e.g., adding, deleting or modifying monitoring demands). Based on a user's monitoring requests, the monitoring system determines the appropriate agents within the hierarchy to be tasked with inspection and evaluation of application events. Monitoring agents are assigned these tasks according to their management role and location. The monitoring system uses fine grain decomposition and allocation mechanisms to ensure that filtering tasks are efficiently distributed among the monitoring agents. Additionally, monitoring agents work collaboratively and concurrently with each other for detection and classification of generated events and forwarding of monitoring information to the appropriate event consumer(s) in a timely fashion. By associating a detected event with a specific action within the monitoring request, users can request that condition-

based activity occur. The resulting action can be an additional monitoring operation or some type of corrective process. Additionally, users may dynamically reconfigure agent operation by manipulating monitoring demands at run-time

The centric component of the monitoring architecture is hierarchical event filtering. The hierarchical filtering approach permits events to be processed and correlated in multiple hierarchical levels in accordance with the complexity of a user's monitoring requests. This significantly contributes to the scalability and performance of the monitoring system. The next section presents, at a high-level, the attributes of the proposed monitoring architecture.

1.5 Contributions

A number of monitoring approaches and systems for monitoring distributed systems have been described as being academic or industrial products (e.g., [10, 16, 18, 27, 32, 36, 44, 46, 57, 58, 60, 68, 69, 78, 80, 82, 90, 93]).

However, these research efforts are each insufficient to meet the requirements of monitoring large-scale distributed systems. Each of these systems/approaches are polarized toward one objective and neglect other equally important requirements resulting in incomplete solutions. Consequently, none of the proposed systems has resulted in a comprehensive monitoring architecture that addresses our objectives or satisfies the requirements of large-scale distributed systems. Scalability is one clear example of this. These systems range from centralized to decentralized but none employ a truly scalable architecture such as that of the Hierarchical Filtering (HiFi) proposed in this thesis. In Chapter 8, these systems are discussed and evaluated with improvements offered by HiFi highlighted.

This work bridges the gap by designing, developing and deploying a monitoring architecture that explicitly addresses the challenges and the requirements imposed by managing large-scale distributed systems. Each component in the overall system is accounted for within this dissertation, from the instrumentation, user subscriptions, event filtering to information dissemination and management reaction. This results in the design and implementation of a comprehensive, scalable, high-performance, dynamic, flexible and non-intrusive monitoring architecture for large-scale distributed systems which is referred

to as the HiFi monitoring system.

Ultimately, we believe that the efficient deployment of Internet/Intranet distributed services have to be accompanied with an effective employment of a monitoring architecture that facilitates dynamic management of these services. Before this can happen, we must understand, build, deploy an evaluate monitoring systems. This dissertation research is one step toward this goal. Our contributions advance the state of the art in monitoring and controlling large-scale distributed systems, networks and services through integration of the following novel techniques:

- **Adaptable Hierarchical Filtering Architecture.** HiFi is the first monitoring system that fully distributes the tasks of detecting and classifying events in an hierarchical fashion. This is accomplished via a set of *monitoring agents.* Only events of interest are allowed to flow up in the hierarchy, thereby reducing perturbation of event propagation in the network and the application environment. Furthermore, this hierarchy is built, and agents are distributed, based on a user's monitoring demands. This hierarchy results in restricting monitoring operations and *intrusiveness* within the observed application entities and reduces intrusiveness imposed on other domains and entities within the application environment. The monitoring hierarchy is dynamic and adaptable to abnormal increases in monitoring load to assure *scalability* and integrity of HiFi in accordance with application needs. The agents management protocol uses a dynamic reliable multicast service (RMS), described in [4], to provide an efficient, dynamic and reliable group communication between agents. This not only significantly improves the scalability and performance of the monitoring architecture but it also provides a reliable management infrastructure in the presence of failures or crashes.

- **Filter Incarnation.** The monitoring model supports *filter incarnation* which is a new scheme that enables users to specify programmable and self-directed monitoring tasks/operations. These tasks permit automatic self-reconfiguration to track system behavior. Users, therefore, can expand their monitoring activities without overwhelming the application environment with too many monitoring requests and sensors. Another very useful application of filter incarnation is dynamic event traces, which enables users to define traces that can be configured during program execution

based on event information. This has a significant impact on improving performance and expressiveness while minimizing the intrusiveness of the monitoring system.

- **Expressive and Declarative Monitoring Language.** The monitoring architecture provides a Monitoring System Language (MSL) which is used to specify the monitoring demands and the application environment. MSL incorporates our monitoring model to integrate detection of primitive and composite events in the same framework. MSL has a unique interface that combines a number of valuable attributes making it a high-level, declarative, expressive, and easy-to-use language. MSL provides a complete interface for specifying all environmental requirements.

- **Dynamic User's Subscription.** The ability to add, delete and modify monitoring requests at run-time can be easily supported in centralized and decentralized monitoring approaches. However, this issue represent a real challenge in a truly distributed monitoring system, such as embodied by the hierarchical filtering-based architecture. In order to support this feature, a number of new algorithms and agent management protocols will be developed to ensure efficient decomposition and distribution of monitoring demands at run-time and to maintain agent consistency in accordance with state changes. Dynamic subscription supports dynamic activation and deactivation for event reporting mechanisms, which reduces generated events and minimizes system intrusiveness.

- **Automatic Instrumentation Utility.** The HiFi monitoring system provides a simple and automatic process for instrumenting program code and for operating the monitoring system via extended system sensors and the automatic agent allocation protocol respectively. HiFi also provides a flexible event reporting mechanism that users can adjust to control the trade-offs of monitoring performance and intrusiveness. Some previous works completely ignored the instrumentation issues, while other support a static, and hand-wired instrumentation procedure that considerably reduces the flexibility and usability of the monitoring system.

- **Adaptive Object-Oriented Filtering Framework.** Research and development of the HiFi monitoring system has facilitated the development of an adaptive Object-Oriented event filtering framework for general-purpose event management applica-

tions [8]. The major contribution of this work is to provide a flexible event filtering framework that can be efficiently adapted to different domain-specific requirements with minimal development effort. In our approach, the event filtering framework captures common components and design patterns of event management in different domains.

- **Comprehensive Monitoring Environment.** Unlike previous work in monitoring distributed systems, HiFi provides a comprehensive and operational prototype that can be deployed for monitoring any large-scale distributed system based on UNIX platforms. Previous work is not fully applicable in monitoring large-scale distributed systems such as [32, 36] as it is either not deployable because of environment restrictions such as use of the Isis system [14, 58], or it is still in a proposal stage. We consider the HiFi achievement as a valuable add-on for the research community to begin understand and analyze the requirements of managing very large-scale distributed systems, such as Internet or MBone services.

The monitoring architecture supports a number of other design features, including the following.

- *Priority-based monitoring:* Events are processed based on associated priorities such that events with higher priorities face minimum monitoring latency.
- *Monitoring space optimization:* The monitoring architecture supports several optimization techniques to reduce or distribute the memory space required for tracking event history.
- *Filtering optimization:* A number of filtering techniques are proposed to reduce the time required to process an event.
- *Dynamic reliable group communication:* Reliable multicasting protocols do not support rapid and dynamic mechanisms for subgroup communications. HiFi work facilitates the development of a new technique for rapid and dynamic group masking [4]. This enables agents to send multicast messages to a subset of its group with minimal group management operations.

The HiFi prototype has been used for monitoring Interactive Remote Instruction (IRI) and for providing effective examples of debugging and system steering applications.

1.6 Dissertation Overview

This dissertation is organized as follows. In Chapter 2, an introduction to monitoring and filtering in distributed systems is presented followed by key criteria in evaluating various mechanisms of event filtering.

Chapter 3 describes the model, language specifications, approach and environment of our monitoring architecture. We explain the terminology used throughout this thesis. Additionally, the chapter describes the specification, examples and features of the monitoring language and describes how it captures the monitoring model. We argue the ability of the monitoring model and language to improve performance and expressiveness while minimizing intrusiveness. This Chapter also describes the potential for integrating HiFi with other existing monitoring tools, such as SNMP [18], to broaden management benefits.

Chapter 4 represents the core of this thesis. This chapter presents alternative filtering techniques used in distributed event management. It then describes and supports the hierarchical filtering-based monitoring architecture explaining the management and communication protocols and the dynamic hierarchical approach. The chapter provides the reader with a tour of the monitoring system, from user specification and processing to action execution. Algorithms and protocols used to implement the architecture (including automatic agent allocation, event and filter decomposition and distribution, dynamic subscription, and event detection) are described. In this Chapter, we also state the impact of these techniques on achieving the dissertation objectives.

In Chapter 5, the following system components and their implementation are described: instrumentation, subscription, filtering and control. For each of these components, the function, subcomponents and design criteria are described. The discussion in this chapter also illustrates design issues and decisions that have a significant impacts on achieving system objectives. An Object-Oriented event filtering framework, for general-purpose event management applications, is also presented.

Chapter 6 presents benchmarking and simulation results for evaluating the perturbation, scalability and throughput of the monitoring system.

In Chapter 7, three examples illustrate the HiFi support for distributed debugging, application steering and general monitoring feedback applications.

Chapter 8 presents a survey and evaluation of related work in monitoring distribut-

ing systems and in event filtering mechanisms. Systems and approaches are compared with HiFi and evaluated based on the LSD systems requirements stated in this chapter.

Finally, in Chapter 9, we conclude by summarizing our contributions, and identifying remaining issues and challenges to be addressed by the HiFi architecture in our future work plan. We also provide references to implementation, documentation and application examples.

CHAPTER II

BACKGROUND

The following background study is divided into two parts. The first part (Section 2.1) addresses monitoring background including the monitoring process and applications while the second one (Section 2.2) discusses event filtering framework. Filtering mechanisms represent the major component of the monitoring system. The survey and evaluation of related work is discussed in Chapter 8.

2.1 Monitoring Distributed Systems: An Introduction

Monitoring is defined as the process of dynamic collection, interpretation and presentation of information concerning objects or software processes under scrutiny [44, 81]. Within distributed systems, monitoring facilitates a variety of tasks including debugging, testing, visualization, animation and systems management. It also provides the foundation for performance management, configuration management, fault and security management and other related activities.

Monitoring Types: Two basis types of monitoring exist. *Time-driven monitoring* acquires periodic status information and provides an instantaneous view monitored object behavior. *Event-driven monitoring* obtains information about events of interest as they occur and provides a dynamic view of system activities based on data collected during those events. Some monitoring systems implement both monitoring types to fulfill varying requirements and constraints [57].

Monitoring Activities: Most monitoring systems perform *four* basic activities:

1. *Generation:* Information about monitored objects is collected to construct a trace which represent an historical view of system activities.

2. *Processing:* Trace information is processed in accordance with required monitoring formats. This may include merging traces, validating information and processes, updating databases, and combining, correlating and filtering captured trace information.

3. *Dissemination:* Processed trace information is forwarded to the request originator, which could include users, managers or monitoring applications.

4. *Presentation:* Processed information is formatted (e.g., text, graphs, etc.) and displayed to the user in the appropriate form.

Monitoring activities are implemented by different monitoring systems specific to system design requirements. Systems may also execute activities in the order appropriate for the design. This results in the four basic activities being executed in a more random fashion, rather than as in a layered architectural approach. For example, information may be displayed without processing or dissemination. Events and reports may be processed solely to generate other events or reports [57].

Monitoring Applications: In general, monitoring is essential to improve the quality of any process (e.g. manufacturing process, management process and production control process). Similarly, in software systems, the process of developing, maintaining and operating distributed applications can be complex. An efficient monitoring of these applications is an essential mechanism to produce good quality applications (e.g. reliable, robust, secure, high-performance). The necessity of monitoring significantly increases when using large-scale distributed systems since they are more susceptible to many problems such as reliability and performance problems. This is because of the large distribution and interactivity of LSD systems which makes it more difficult to debug and steer. In this section, we will present a number of monitoring applications which are important to improve the quality of distributed systems. We also show how monitoring is more compelling for LSD systems.

- *Debugging and Testing:* Unlike centralized or isolated systems, bugs or incorrect

behavior in LSD systems may be related to multiple components distributed across the application environment. In addition, related events can be generated simultaneously. In this environment, it would be difficult or may be infeasible for the system developers to track the state of the system and collect information on the run-time functional behavior of the system for debugging and testing purposes. The proposed monitoring architecture enables the developers to debug and test LSD systems by (1) detecting event of errornous or incorrect operations, (2) requesting activity reports of certain functions or components, and (3) producing event traces of the entire application history within a specific time-interval. Events are detected and forwarded by the monitoring system at run-time regardless of the events locations (i.e. remote or local machine) and how complex the events are. A complex event is composed of set of events that are generated from different sources or components of LSD systems. For example, the error of the *receiving* message size being different from what has been sent is a complex event since it is distributed in the application environment (senders and receivers).

- *Performance Tuning:* The environment of LSD systems (i.e. Intranets or Internet) changes dynamically due to the variable load on the system and the network. For example, a congested link at this moment may not be congested after sometime. LSD systems need to adapt to the changes in the network or system status in order to maintain a good performance during its execution. This may support improving or maintaining the quality of the services (e.g. fast response) provided by LSD applications and thereby meet the users/customers satisfaction. The monitoring system is an effective means for performance tuning and application steering. The management decisions of the monitoring applications (e.g. tuning) may be based on set of correlated events which are concurrent and distributed in the LSD environment. Therefore, the *reactive control* monitoring application of the LSD system should be able to request the monitoring of specific local or global events that could effect the performance of the application and adjust the application control parameters to adapt to the new conditions. For example, it may be desirable to dynamically adjust the sending rate in multicast group to the average of the the receiver rates (instead of going with the slowest receiver as most reliable multicast protocol do). The re-

active control module may get a continues feedback on this global event or it may get informed only when the average of the receivers rate goes beyond a threshold values. In either case, the monitoring system is essential for providing such information at run-time which can be used (by reactive control components, for example) subsequently for control decisions.

- *Fault Recovery:* Faults occur during the execution of LSD systems because of problems in the environment (e.g. wrong system or network configuration), software bugs or improper user operations. It is important for the application developers and the system managers to know the source of any failure in order to improve the robustness and the reliability of the application [43, 91]. The proposed monitoring architecture can be used effectively to classify and report all failures during the application execution so recovery procedure can be manually initiated. Furthermore, the proposed monitoring architecture supports an automatic fault recovery service where corresponding recovery procedures are initiated automatically if a failure detected. Therefore, the monitoring architecture provides a centralized control of application failures that are distributed in LSD environment.

- *Security:* The monitoring system can be used to detect and report security violation events such as repeated illegal logins or attempts of unauthorized file accesses. The monitoring mechanisms identifies these events based on specific pattern or set of values revealed by the application itself.

- *Correctness Checking:* The monitoring architecture can be used as a verification technique to ensure the consistency with a formal specification. The feedback information on the run-time behavior supplied by the monitoring is analyzed by software verification tools to discover any inconsistency.

- *Performance Evaluation:* The monitoring technique can also be used to evaluate the applications performance at run-time. The monitoring mechanism is used to extract data from the application during its execution which is later analyzed to assess system performance. Usually, such monitoring techniques require some hardware support to assure accuracy and efficiency [36].

Although, the focus in this thesis is on the first three monitoring applications (debugging,

performance tuning and fault recovery), the monitoring architecture can potentially be used in other monitoring applications.

2.2 Event Filtering: Key Criteria and Design Trade-off

The event filtering mechanism is the core component of the monitoring architecture. It serves as an efficient mechanism for detecting (and rejecting) generated events (primitive or composite). Event filtering also reduces high volumes of event traffic as monitoring work is subsequently offloaded from network and the consumers hosts. Thus, event filtering has a significant impact on the performance and scalability of the monitoring architecture. In this part of the thesis, focus is given to major issues related to designing event filtering mechanisms. An evaluation of existing filtering techniques, based on our design objectives, is described in Chapter 8. In order to achieve the most efficient and flexible design and implementation of event filtering, we have explored the event filtering mechanism in various application domains beside the system monitoring and management application. Event filtering is useful in several domains including distributed systems toolkits, network and system management, communication protocols, and active databases. In this section, we device the design framework of event filtering mechanism by identifying the key criteria and design trade-offs of each one. We also classify the existing event filtering systems based on the key design criteria which helps evaluate alternatives techniques of event filtering mechanisms.

Application Domain

Event filtering is used as a classification mechanism in several application domains. Each domain uses filtering for different purposes according to domain-specific goals and requirements.

• **Distributed Systems Toolkits:** Event filters are used in distributed systems for *validating* incoming messages. Filters are used to distinguish (classify) invalid messages (e.g. errornous or unauthenticated messages) by passing it through a series of validation filters [14].

• **Network and System Management:** *Monitoring* is one of the most common applications of event filters in network and system management environments. Using filters, a network/system administrator can classify and analyze certain types of events and collect statistical information about different aspects of network or system operation. Examples of events filtering toolkits in the network and system management domain are the Packet Monitoring Program (PMP) [16], HP OpenView [45], snoop [87] and `tcpdump` [40] in the UNIX environment.

• **Communication Protocols:** In operating systems, packet filters are used as an efficient technique to *demultiplex* incoming packets and forward them to the corresponding communication endpoints(e.g., [12], [59], [63], and [94]).

• **Active Databases:** Event filters are used in active databases to construct *triggers*. Triggers are specified as *event-condition-action* tuples (i.e., performing an action if the event occurred and condition is satisfied). Application examples of active databases systems includes *financial applications* such as stock market (e.g., [27] and [29]).

Event Filter Internal Representation

The internal representation of filters is a key issue in designing and evaluating event filtering mechanisms. The internal representation determines the *structure* (data structure) and the *operation* (algorithm) of the filtering mechanism. The internal representation of a filter has a major impact on *performance, scalability* and *functionality* of the filtering mechanism. In the following, we classify event filtering mechanisms based on the functionality of their internal representation. For each classification, we present alternative internal representation models (data structures and algorithms). To focus the discussion, we show a filter example and how it is constructed using each representation. This filter example captures all packets with an IP *source address* "foo" and either the IP *destination address* is "bar" or the TCP *destination port* is "ftp."

(1) Primitive Event Classifiers: The internal representation of this type of filter is usually implemented in the operating system kernel to support efficient classification of primitive events. In particular, it does not record any history of events detected in the

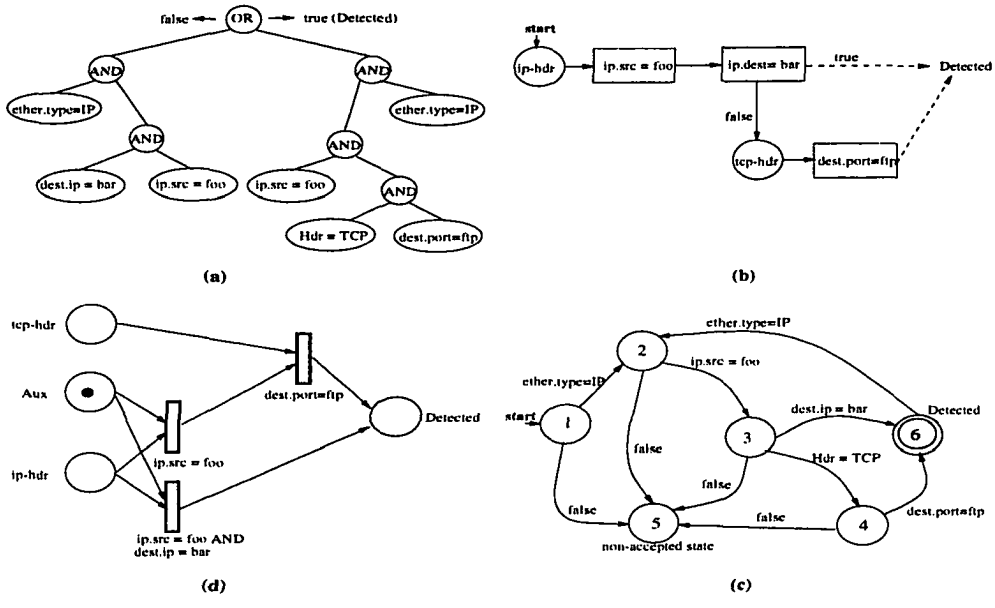


Fig. 2.1. Filter Internal Representations. (a) Boolean Expression Tree. (b) Directed Acyclic Graph. (c) Deterministic Finite Automata. (d) Petri Nets.

system. Thus, the primitive event filters are unable to classify composite or global event. Conventional packet filters [12, 59, 63, 94] are examples of this type of filter. Packet filters primarily classify and demultiplex communication protocol packets to user process endpoints. The remainder of this section presents alternative internal representations of primitive event filters:

• **Boolean Expression Tree Representation:** A boolean expression tree is represented as a binary tree. Each interior node in the tree represents a boolean operation (e.g. `AND`, `OR`). The leaves represent test predicates (also called *masks* or *cells*) on event fields. Each edge in the tree connects the operator (parent node) with its operand (child node).

The algorithm for manipulating a boolean expression tree is based on a "bottom-up" parse of the tree. Events are classified by evaluating the tree starting at the leaves (test predicates) and propagating the results up to the binary operator at the root. The event is matched if the root of the tree evaluates to "true". Figure 2.1-a illustrates an

TABLE 2.1

EVENT FILTERING INTERNAL REPRESENTATION

Classification	Events Generated		Internal Representation		
	Location	Format	Node	Graph	Algorithm
Primitive Event Classifiers	remote	notification messages	predicate or operators	acyclic	DAG or Tree
Composite Event Classifiers	local	exceptions	event	cyclic	DFA or PN

example of this representation model.

• **Directed Acyclic Graph (DAG) Representation:** A DAG representation is implemented as an acyclic graph (Figure 2.1-b). Its nodes represent the test predicates and the edges represent the control transfer. The DAG is parsed top-down such that if the test predicate (also called a *cell*) is true, the right-hand edge is traversed, otherwise the left-hand edge is traversed. Thus, the evaluation result of the test predicate (either *true* or *false*) determines the edge to traverse. An event is matched if the terminating node (leaf node) is denoted as *true*. There are two terminal nodes in the graph, the *true* node that denotes the acceptance of the event and the *false* node that denotes the event rejection.

(2) Composite Event Classifiers: The internal representation of this type of filter is usually implemented in the user-space to detect and classify composite events, as well as primitive events. An example of this type of filter is the event filtering mechanism used in active database systems. The main function of the internal representation of these filters is to track events detected in the system, classify composite events as they are recognized, and trigger actions based on the detected events. In this environment, the primitive events are the basic database operations performed in the system (such as add and delete queries) which are identified locally by the filtering system as a programming language exceptions [19]. Therefore, unlike the primitive event filters, the composite event filters does not support means to classify primitive events that are represented as notification messages. In other words, each primitive event is identified by the exception handler and a mechanism for parsing notification messages is not required in this environment to

determine the event identity. The following examines alternative internal representations for composite event filters.

• **Deterministic Finite Automata (DFA) Representation:** DFA representation is a finite state machine graph. A transition between two states represents an event occurrence. Each state represents the history of the system environment either before or after the occurrence of an event. For example, if event x occurs, a transition on x from one state (h_1) to the next state (h_2) occurs. In this case, h_1 and h_2 represent the environment before and after x occurs, respectively [29].

In this model, a filter consisting of a single primitive event is represented by a three state automaton consisting of a start state, accept state, and non-acceptance state. From all states, the transition on event x (the event to be detected) triggers a transition to the accept state. Otherwise, on all other events the transition is to the non-acceptance state.

A filter consisting of composite events is constructed by combining the DFAs of primitive events together into one DFA using the joining rules of Finite Automaton. By definition, all DFA transitions are deterministic. An example of this representation model is given in Figure 2.1-c.

• **Petri Nets (PN) Representation:** A model of Petri nets called *Colored Petri Nets* (CPN) has been used in active database systems [27] to construct event filtering mechanism. The following describes this model:

All predicates are represented as states called *places*. CPN has a number of *tokens* assigned to places. If a place has a token, it is called a *marked* place. A *place* is marked when a predicate of that place is matched. Operations between places (predicates) are represented by *guard functions* that are checked if all associated places are marked.

Whenever a predicate match occurs, the input *place* of the CPN is marked with a token. Initially, *tokens* are stored in auxiliary places of the CPN to designate the marking at creation time. Now, if *all* input places of a state are marked, the *event variables* that are denoted as labels in the CPN arc are bound to the value of the appropriate token and the *guard function* is evaluated. If the *guard function* evaluates to true, the state *transition* is fired, the token is transferred to the output place of the transition, and the variable value is propagated to the next state. More details of Petri Net concepts and behavior can be

found in [27]. Figure 2.1-d illustrates the same filter example modeled by a CPN.
To conclude with, both primitive and composite event classification functions are necessary for monitoring LSD applications. Primitive event classification is required since events in LSD applications are represented as notification messages. In addition, composite event classification is required since the event filter may need to track event history in the system. Table 2.1 compares these two types of event classifiers. Each of the previous event filtering techniques lack this generality and functionality.

Event Filter Programming Interface

An *event filter programming interface (FPI)* provides a language for defining filter components (such as filter expressions and actions). An event filter expression describes all *predicates* involved in the event definition (including the message fields and the operators). The *actions* describe what will be done when the desired event is detected. Forwarding the detected event to a corresponding consumer could be an example of a filter action. The filter definition is used by FPI to construct the filter internal representations discussed in this section.The internal representation is then used to operate the constructed filter definition. The *expressiveness* (the expressive power of the event filtering definition) and *ease of use* are two major trade-offs in designing FPI.

This section describes various ways to define the syntax of event filters. A filter definition can be programmed at different levels of abstraction. In the following we classify the filter programming interfaces according to its level of abstraction and indicate examples for each one (detailed discussion of these examples with filter program samples can be found in [2]). Table 2.2 summarizes the material discussed in this section.

- **Imperative Low-level Programming Interface:** Low-level languages/interpreters (such as assembly or micro-code languages) have been used to program filters imperatively. This programming interface is imperative since the filter definition is given as a program that describes the semantics of the filter operation (i.e. how the predicates get evaluated against the notification fields). The *Stack-based Interpreter* [63] and *Register-based Assembly Language* [59] are examples of the communication protocols packet filters that use this level of abstraction.
- **Imperative High-level Programming Interface:** A high-level description language (similar to high-level programming languages) has been used to define filters. This pro-

TABLE 2.2

EVENT FILTER PROGRAMMING INTERFACE DIMENSIONS

Filter Interface	Application Domain	Examples	Design Issues	
			Programming	Tools
Imperative Low-level	communication protocols	CSPF, BPF MPF	Assembly Languages	Interpreters
Imperative High-level	network management	IPM	High-level Languages	Interpreters
Declarative	comm protocols	PathFinder	Special Script	Interpreters
High-level	active databases	SAMOS, Ode	OODB	Compilers

gramming interface uses the imperative approach since a high-level language describes the exact semantics of filtering operation. The *Interpretive Pseudo-Machine (IPM)*[16] is an example of this type of filter programming interface.

• **Declarative High-level Programming interface:** Event filters may be specified in a high-level declarative language. The previous event filter programming interfaces use imperative approach where filter definitions are given as programs. Other event filtering interfaces use a declarative programming interface where filter definitions are given by a pattern (event) match [12, 93]. Examples of declarative filter programming interfaces includes *Rule and Database Languages* [93] such as SQL embedded with Prolog and *Declarative Scripting Language* [12] which is a customized script language used for packet classification.

Models of Event Filtering

A model of an event filtering mechanism determines the structure of event filter components such as (1) predicates, (2) event definition, and (3) filter expression. The filter model determines the expressiveness power of the event filtering mechanism. In this section, we will highlight some of the major issues that impact the model of the event filter. More details can be found in [2]. Table 2.3 summarizes our discussion.

Event Definitions: The definitions of events to be detected must be described concisely

TABLE 2.3

MODELS OF EVENT FILTERING

Event Definition	Primitive events	DB operations	retrieve, update
		DB and patterns	pattern match in DB
		Binary operations	set of test conditions
	Composite events	No support	use primitive events only
		Supported	combined primitive events
Filter Expression Definitions	Operators	Basic	such as `OR`,`And`, `Not`
		Advanced	from the basic ones
	Parameters	Static	fixed values
		Dynamic	modifiable at run-time
	Time Intervals	No support	no time information
		Primitive	explicit event time creation
		Advanced	implicit event time creation

in a filter program. An event can either be a *primitive* or a *composite* event (which is constructed from one or more primitive events). Hence, modeling the event filter requires defining primitive and composite events definitions. Various definitions have been used in the existing filtering techniques. For example, in the communication protocol event filters [12, 16, 59, 63, 94], events are primitive which consist of set of binary predicates forming a boolean expression. Composite events are not supported. On the other hand, in active database [27, 29], primitive event are database operations such as update record which is similar to a function call (not notification-based) and a composite event is just a combination of primitive events.

Filter Expression Definitions: An event filtering expression defines the relation (*operators*) between basic elements of an event filter: *relational predicates* (e.g. ($17 \leq$ *transaction_id* < 40) or event designators (e.g. `Overloaded(foo)` such that "foo" is a machine name). Filter expression enables programmers to develop more complex event filters by comprising large number of predicates and/or event designators in the filter expression definition. The following factors determine the efficiency of the filter expression:

TABLE 2.4

KEY CRITERIA OF EVENT FILTERING

Key Criteria	Definition	Design Feature
Application Domain	class of applications	determines functionality
Internal Representation	structure and algorithms	performance, scalability
Programming Interface	programming language	ease of use
Models of Event Filtering	filter components	generality and flexibility

- **Filter Expression Operators:** A filter expression can be either an expression of predicates or an expression of event designators. The primitive event classifiers use the former, the composite event classifiers use the latter (includes database operations). In both cases, special operators are required to join the predicates in a filter expression. There are two kinds of event expression operators (1)*basic operators* which are simple logical operators such as `AND`, `OR` and (2)*advanced operators* which are usually derived from the basic operators.

- **Parameterized Filter Expression:** Predicates in a filter expression consist of one or more parameters that are used to analyze and compare against a message fields. Examples of parameters are `message_id` and `transaction_id`. Some event filters [59, 63, 94] have a *static parameters* whose values can not be altered after its initialization. Other event filters [12, 27, 29, 93] have *dynamic parameters* whose values can be changed dynamically at run-time and during the filtering operation. Examples are illustrated in [2].

- **Time-Intervals in Event Filtering:** Some applications require classifying events based on time-interval functions (such event creation time and event temporal ordering). Moreover, some applications require detection of temporal events*. Thus, monitoring time-intervals may be needed to filter events. In this case, defining time and interval functions must be supported in the event filter definitions. Some event

*events occur according to specified time-precedence.

filters [12, 59, 63, 94] have no support of this feature, some [29, 93] have a primitive support such as providing timestamps in the events and others [27] support advanced time-functions such as (*Event_A* `before` *Event_B*) filter.

To summarize our discussion on the key criteria of event filtering, Table 2.4 outlines the key criteria and shows the related design feature influenced by each corresponding criteria.

CHAPTER III

DESIGN APPROACH

We start presenting the monitoring architecture by describing the monitoring model, language and environment. These three issues represent the basics of the monitoring architecture because they define the monitoring dynamics, the system interface and the system interaction, respectively. We introduce the terminology used throughout this thesis. This chapter also describes the specifications, features and examples of the monitoring language and describes how it captures the monitoring model. We then argue that the monitoring model and language are capable of improving the performance and expressiveness while minimizing intrusiveness of the monitoring system.

3.1 Monitoring Model

In order to present a complete abstraction of the monitoring problem, our work must include modeling the *application behavior*, the *monitoring demands*, and the *monitoring mechanism* when considering the design objectives presented in Section 1.2. In this section, we present our model of the monitoring process and introduce the terminology that we use throughout our discussion in this thesis.

3.1.1 Event-based Abstraction of Application Behavior

The program behavior can be expressed in a set of events revealed by the application during execution. In our monitoring model, we call the monitored programs *event producers* which continuously emit *events* that express the execution status. An *event* is a significant occurrence in LSD systems that is represented by a *notification message*. A notification message typically contains information that captures event characteristics such

as event type, event values, event generation time, event source, and state changes. *Event signaling* is the process of generating and reporting an event notification. For simplicity, we use "event" and "notification" interchangeably in this thesis. That is, we consider a notification to represent an event. We also classify two types of events used in our model: *primitive events* which are based on a single notification message, and *composite events* which depend on more than one notification message. In other words, the composite event represents a logical relation of two events or more to form a higher abstraction of the program behavior. For example, *error events* generated by producer (i.e. application entity) "foo" are primitive events since they can be detected by checking the fields of a single notification that has source address "foo" and event type "Error". On the other hand, in order to discover that two producers (or more) are generating error messages (event type field is "Error"), the notification `event type` field of at least two different producers must match the value "Error". Therefore, we call this event a composite event because detecting this event requires the matching of multiple primitive events of different producers. We will use the term *event pattern* to refer to any set of related primitive or composite events which may represent a program behavior. In our model, the event format (notification) is a variable sequence of *event attributes* determined by the user but it has a fixed header used in the monitoring process. An event attribute is a predicate that contains the *attribute name* which typically represents a variable in the producer (i.e, program) and a value. The event format also determines the type of event signaling: *Immediate* to forward the generated event immediately, or *Delayed* to allow buffering/batching events in the producer before sending them. Table 3.2 shows the formal definition of the event in BNF [70]. This event abstraction enables consumers (1) to specify any arbitrary event format in a declarative way, and (2) to construct a complex (multi-level) abstraction of a program behavior using composite events. In addition, the event abstraction enables the consumers/users to assign values to the event attributes and does not require specifying attribute type (e.g., `int, float or string`). This is unlike the CORBA IDL abstraction which requires each attribute type to be determined and does not permit the user to specify values in the data constructors [66].

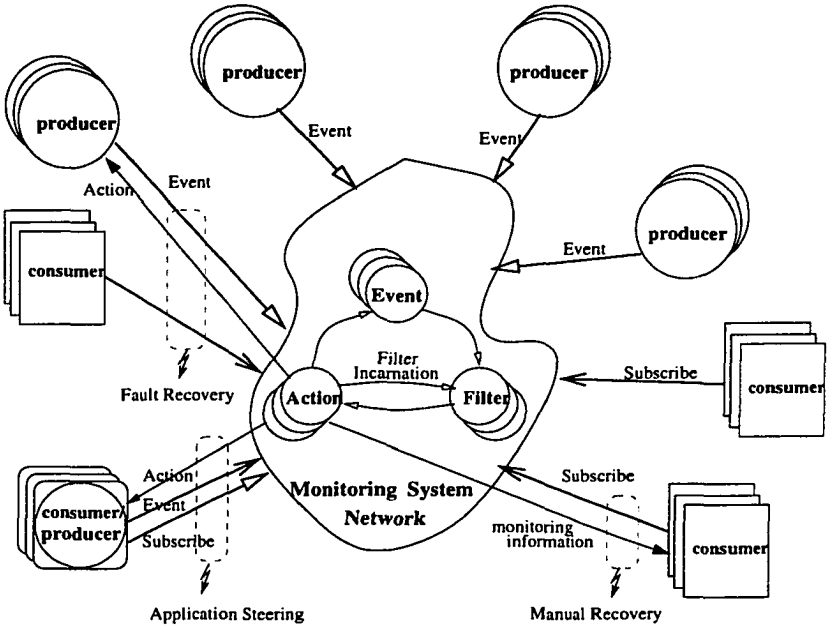


Fig. 3.1. Monitoring Model.

3.1.2 Filter-based Abstraction of Monitoring Demands

We call the monitoring objects (e.g., human or software programs) *event consumers* since they receive and present the forwarded monitoring information. The consumers specify their monitoring demands via sending a *filter* program via the *subscription process* which configures the monitoring system accordingly (see Figure 3.1). A filter is a set of *predicates* where each predicate is defined as a boolean-valued expression that returns *true* or *false*. Predicates may be joined by *logical operators* (such as `AND`, `OR` and `NOT`) to form an expression. In our model, the filter consists of three major components: the *event expression* which specifies the relation between the interesting event, *filter expression* which specifies the attributes value or the relation between the attributes of different events, and the *action* to be performed if both event and filters expressions are *true*. Table 3.1 shows the formal definition of the filter in BNF. As the event abstraction emphasizes the declarative aspect of the model, the filter abstraction improves the expressive power and the usability of the monitoring system. For instance, the filter program not only enables

TABLE 3.1

BNF OF HIGH-LEVEL FILTER SPECIFICATION LANGUAGE

<Filter> ::= **FILTER** = <Filter_Body>
<Filter_Body> ::= [<Event_Expr>]; [<Filter_Expr>]; [<Actions>];
<Filter_Name>.
<Event_Expr> ::= (<Event_Name> <Event_Op> <Event_Expr>)
| <Event_Name>
<Filter_Expr> ::= (<Predicate> <Filter_Op> <Filter_Expr>)
| <Predicate> | **TRUE**
<Predicate> ::= (<Pred_Att> <Relation> <Pred_Att>)
| (<Pred_Att> <Relation> <Value>)
<Pred_Att> ::= <Event_Name>.<Att_Name>
<Filter_Op> ::= <Event_Op>
<Actions> ::= <Action> ; <Actions>
<Filter_Name> ::= <Program_Name> ::= <String>

---

users to describe the relation between the events of interests, but it also permits users to specify the relation between the attributes of different events. In addition, consumers can overload the attributes values specified in the event specification by assigning new values in the filter expression which enables creating different instances of the same event in the different filters. The action component is used by consumers to support reactive control services, such as fault recovery and application steering, which is a target application of the monitoring architecture. The process of coalescing two or more filters together into one global filter structure inside the monitoring system is called a *filter composition*. The filter composition takes a place when more than one filter is submitted from one or more users in the same monitoring environment.

3.1.3 Event-Subscription-based Monitoring Model

The monitoring model must be capable of supporting the target applications and the objectives of the monitoring architecture. In the above discussion, event abstraction and

filter abstraction determine the inputs to the monitoring system. Figure 3.1 shows the basic state diagram of the monitoring model. The producer behavior is observed based on the event generated (event-based) and on the monitoring requests (subscription-based). Events received in the monitoring system are classified based on existing filters. If an event is detected, the action specified in the filter is performed such as forwarding the monitoring information to the corresponding consumers. Thus, our monitoring can be viewed as an *event-demand-driven* model. It is event-driven, as opposed to a time-driven model, because monitoring information is reported based on event occurrences. Also, it is demand-driven, as opposed to a trace-driven model, because generated events are filtered and actions are performed based on the consumers' demands. Actions in the monitoring model can be simply *executing a program* (local or remote) or *forwarding* the detected event to the corresponding consumers which are both necessary for reactive control (automatic fault recovery and application steering) and distributed debugging, respectively.

In order to improve the dynamism and the expressive power of the monitoring system, the model provides more complex actions: a new event or a filter incarnation. Generating new events as an action has significant impact in improving the expressive power, performance and usability of the monitoring system as follows: (1) The event-filter-action cycle (see Figure 3.1) enables the consumer to activate a series (loop) of monitoring operations (filters and actions) automatically without having him/her to intervene in the monitoring process. For example, a failure may occur in a producer (process) as result of abnormal close of communication connection (primitive event). In this case, the efficient management operation involves *failure recovery* as well as *sending* an event to further diagnose the process (producer) that closes the connection. In addition, this action-event may trigger other filters to check the status of other running processes. Based on this, new actions (e.g., recovery procedures) could be performed, (2) An action could generate a "summary" event which summarizes the information of detecting a composite event (e.g., the event expression consists of multiple events). This enables suppressing the information of multiple events into one event (summary event), thereby avoiding event report implosion and reducing the event traffic, and (3) Performing an action such as executing a program may change the state of a running program. Therefore, sending an event that reveals the state change to the monitoring system is important to allow re-observing the behavior, thereby enabling automated application steering.

An action can also be a filter manipulation (typically, adding a new filter, deleting a filter, and modifying a filter). For example, another new filter can be activated in the monitoring environment as a result of detecting an event. We call this *filter incarnation* (see Table 3.1 and Figure 3.1) because a filter may add, delete or modify a new filter in the system. The filter incarnation enables *dynamic monitoring* which allows the monitoring system to re-configure itself automatically based on event occurrences. This feature is important because it enables the consumer to control the monitoring granularity, and thereby minimizing its intrusiveness. In particular, the consumers can subscribe for a small number of filters, however, these filters may activate other filters when a specific event pattern is detected. For example, assume a consumer wants to start monitoring the "transmission" events of a program (producer) X only when the *drop rate* of the "receiving" event of a program (producer) Y exceeds a certain threshold. Then, the consumer can specify one filter that monitors the "receiving" events of Y which will trigger another filter to monitor the "transmission" events of X if the drop rate exceeds the threshold. This permits activating the transmission status filter automatically and at the proper time which minimizes the monitoring perturbation in the application environment. Therefore, the monitoring model enables dynamically activating/deactivating the appropriate monitoring operations (or filters) at the right time (event), and thereby relieving the system environment from the overhead of launching multiple filters or monitoring requests simultaneously. Moreover, the filter incarnation feature provides an extendible programming environment utilizing the power of the recursive event-filter-action model. Table 3.5 shows the formal definition of the monitor action in BNF. Section 3.3 presents a detailed discussion, examples and features about the monitoring language specifications.

3.2 Integrated Application-Level Software Monitoring

Our monitoring approach is strictly a software monitoring system. Unlike hardware [34] and hybrid monitoring [36], no special hardware is required in order to use the monitoring system. This design attribute is important to assure portability and flexibility of the monitoring system in different platforms and to minimize cost. On the other hand, monitoring intrusiveness and resource sharing are given a substantial consideration in the architecture design as will be shown later in this thesis. The HiFi monitoring system is an application-

level program that interacts with event producers and consumers through a well-defined interfaces. The communication with events producers is almost unidirectional since the events flow from producers to the monitoring system, and communication in the other direction is very controlled and minimal (see Chapter 5). This is important to minimize program perturbation. Consumers send filters and receive forwarded notifications to and from the monitoring system, respectively. In Chapter 4, we will discuss the type of communication channels used in both cases. Moreover, implementing the monitoring system in *user-space* (application-level), provides flexibility for testing and debugging, portability, system protection (security) and dynamic linking. On the other hand, such advantages can not be obtained in kernel-level monitoring systems such as [87].

Another advantage of using application-level monitoring is the ability to interact with other available monitoring tools such as SNMP [18] and CMIP [82] or non-standard management utilities such as `top` and `perfmeter`. Although the monitoring architecture is an event-based, it can communicate with other monitoring tools via *event emulation layer* which acts as a proxy agent between the monitoring system and other monitoring tools. The main function of the event emulation layer is to mimic the event producer interaction by requesting and collecting the monitoring information from the external tools, and providing it in event-based format to the monitoring architecture. The event emulation layer consists of two parts: *tool-independent interface* which is common for all tools that performs event signaling based on our monitoring model described above, and *tool-dependent interface* which is peculiar to the external monitoring tool. For example, the event emulation layer frequently polls an SNMP agent, analyzes collected events and generates notifications if the pre-defined primitive events by the consumer are detected. This architecture enable the monitoring model to correlate events from the producer with other events (such as CPU utilization or memory usage) from external management tools. This integrated monitoring environment is very beneficial for understanding the program behavior and for fault diagnosis [15].

3.3 Monitoring Language

This chapter discusses the main issues involved with designing a monitoring system as part of the overall system architecture. Major components of the monitoring system,

including the monitoring programming language and the monitoring language/interface, are described with design considerations and trade-offs. Critical attention must be given to the monitoring programming language as this is the access point used by consumers for controlling the monitoring system.

3.3.1 Design Principles

A monitoring system must provide a user interface to permit configuration of the system to desired monitoring specifications. Because of the complexity of these configuration tasks, the interface must provide flexibility beyond that of a traditional API which supports only service requests (e.g., function calls). To provide this flexibility, the monitoring systems must provide a *programming interface* which is referred to as the *monitoring system language (MSL).*

The MSL provides a means for defining event formats (e.g., event attributes), filter components (e.g., filter expressions) and the application environment. Based on which event patterns are detected, the monitoring system uses information defined in MSL to construct the corresponding *filtering internal representation.* The filtering internal representation is discussed later in this section.

The characteristics and quality of the monitoring programming language is influenced by a variety of design alternatives [53]. Key characteristics of language include:

- *Expressiveness* – the expressive power of the monitoring language depends on the *abstract model* of the monitoring mechanism, and the availability of a rich set of *expression operators* provided by the language itself. The abstract model determines the capability of the monitoring system to represent and execute user demands/requests. Flexibility in constructing filtering expressions is achieved by providing a rich set of operators.

  Monitoring systems that lack expressiveness have limited capability to define complex monitoring demands. As a result, this restricts the use of the monitoring system within a particular application domain (application-dependent). Examples of such restricted monitoring languages are presented in [2].

- *Ease of use* – Some event filter programming interfaces are declarative languages where filter definition is given as a pattern/event match. This allows specification of

events of interest without focusing on low-level programming details. In a declarative design, there is no need to specify program control and data structure operations as is the case with imperative languages. Other programming interfaces, such as those which utilize "assembly language", are more imperative in nature. This requires users to deal with low-level details (such as message formatting and bit/byte operations).

3.3.2 Language Design Trade-offs

We classify existing event filter programming interfaces into low-level and high-level interfaces according to the abstraction level, and into imperative and declarative interfaces according to the programming approach. Here, we describe number of trade-offs that arise when designing and implementing event monitoring/filtering programming interfaces.

Low-level vs. High-level: Low-level interfaces, such as assembly language, often perform more efficiently than high-level interfaces. However, high-level programming filters increases usability since they are easier to program. High-level filters are also more portable since they are less dependent upon hardware and underlying internal filter representation.

Imperative vs. Declarative: The declarative approach makes filter programs more concise and easier to write when compared with the imperative approach. Thus, the declarative approach increases usability, extensibility and maintainability of the monitoring programs. However, declarative languages require programming within the language framework. As a result, the declarative approach imposes some limitations that may decrease the expressiveness of event filter programming. For example, declarative filtering interfaces such as PathFinder [12] are customized to work for specific applications (like demultiplexing TCP/IP packets). In contrast, the imperative approach overcomes this limitation through the use of language constructors which permit more expressive filtering programs. An example of a filter imperative language interface is the Interpretive Pseudo Machine (IPM) described in Section 8.2. Designing a declarative and expressive monitoring language is a challenging issue of which this section attempts to address.

Basic Operators vs. Advanced Operators: Some filter programming interfaces provide basic operators (such as `AND`, `OR`, and `NOT`), while others provide more advanced operators such as `Before`, `After`, and `Sequence`. Advanced operators increase the expressive power of event filtering expressions. There are two disadvantages of using advanced operators: (1) the increased performance overhead at run-time due to interpretation and processing of these operators, and (2) the increase complexity of use. Conversely, basic operators are usually simplistic as they represent the core instructions of the filter expression. Therefore, basic operators typically incur less run-time overhead and are easier to use.

Interpreters vs. Compilers:

- Interpreters are normally used when filters are implemented in the OS kernel. In this type of implementation, interpreters provide better system protection and robustness than compilers.
- Compilers are more convenient when the filtering mechanism is implemented in user-level applications. Compilers permit dynamic linking and run-time optimization thereby increasing run-time efficiency of event filtering mechanisms.
- Interpreters continuously re-examine program code increasing execution overhead and causing significant degradation in monitoring performance.
- Interpreters may also have greater core storage requirements when compared with storage needs for compilers. The interpreter and supported routines usually must be kept in memory simultaneously using larger amounts of core storage resources. In contrast, compilers dynamically link to target routines at run-time, which minimizes space utilization.

3.3.3 Event Specifications

An event is defined as a significant occurrence during program execution. Events are represented by notification messages sent from the application to the monitoring system. The notification message encapsulates all information needed to identify a specific event. This implies that there is a one-to-one correspondence between the event names and the

TABLE 3.2

BNF OF HIGH-LEVEL EVENT SPECIFICATION LANGUAGE

<Event> ::= **EVENT** = <Event_Body>.
<Event_Body> ::= <Prim_Event> | <Comp_Event>
<Prim_Event> ::= {<Fix_Att> ; <Var_Att>} <Event_Name>
<Comp_Event> ::= (<Prim_Event> <Event_Op> <Comp_Event>) |
(<Prim_Event> <Event_Op> <Prim_Event>)
<Fix_Att> ::= **ModuleName** = <String>,
FuncName = <*String*>, <*Report_Mode*>
<Report_Mode> ::= **Immediate** | **Delayed**
<Var_Att> ::= <Predicate> , <Var_Att> | <Predicate>
<Predicate> ::= <Att_Name> <Relation> <Value>
<Event_Op> ::= $\wedge$ | $\vee$ | $\sim$
<Relation> ::= < | > | = | $\neq$ | $\leq$ | $\geq$
<Value> ::= <Number> | <String>
<Event_Name> ::= <Att_Name> ::= <String>

---

notifications. In other words, no two events of the same name (identity) have the same notification. In our discussion, we use words "event" and "notification" interchangeably.

The *High-level Event Specification Language (HESL)* is the part of the MSL used to define target events. Table 3.2 shows the BNF of the HESL. In the HESL, an event must have the `EVENT` construct as a prefix and the event name (event identification) as a postfix. The event name (`Event_Name`) must be unique within a single application environment. The MSL provides constructs for defining composite and primitive events specifications. A composite event consists of a set of event pairs (actually primitive or composite event names) that are connected relations `Event_Op` (*AND, OR and NOT*). Therefore, a composite event may consist of primitive and/or composite events other than the event itself. In addition, all event names included in the composite event definition must be specified before the definition of the composite event. Primitive events consist of set of predicates that, in turn, consist of an attribute, an attributes value, and a logical

relation (e.g., $<, =, >$). For example, `IPDest = 224.5.5.5` represents a predicate of an event attribute `IPDest` (destination IP address) whose value must equal the IP address `224.5.5.5`. As another example, the predicate `Nacks > 100` indicates the value of the "number of negative acknowledgments" (`Nacks`) attribute for this event must be greater than 100.

Primitive events have *fixed* attributes: program/process name (`ModuleName`), function/procedure name `FuncName`, and the reporting mode (`Report_mode`). The event notification must specify the values of these three attributes otherwise reported events are rejected. The reporting mode can be either *Immediate*, which generates the event and sends it to the monitoring system right after its occurrence, or *Delayed*, which uses a batch mode to buffer events until one of the following conditions occurs: (1) an immediate event occurs, (2) the program invokes the `FlushEvents()` which is a service supported by the instrumentation routines, or (3) the number of buffered events exceeds the maximum threshold allowed in the monitoring agent. A fourth condition, set by time limits, is purposefully avoided to prevent perturbations caused by use of the timer interrupt. However, consumers/programmers can still set up timers in the program and flush the event buffer using `FlushEvents()` service. This issue is discussed further in Section 5.1. The reporting mode enables users to control monitoring intrusiveness and to manage the level of *event freshness*. Monitoring intrusiveness can be minimized by reducing the events reporting rate when batch mode is used. Event freshness, which is the elapsed time between event occurrence and event reporting times, can be reduced to minimal if immediate mode is used, or reduced to a certain limit if `FlushEvents()` is frequently invoked. Furthermore, the report mode can also be used to assign an event priority class number which is an integer used by the monitoring system to discriminate between events. Events of smaller report mode numbers have more priority than those of larger numbers.

The next part of the primitive event attributes is *variable* (see Table 3.2). The variable attributes part permit the user to define an arbitrary list of attributes to describe desired events as in the examples shown below. However, since these events are originally emitted by the program itself, the event attribute values must be associated with the program variables in order to reflect the internal state of the program execution. Therefore, in the HESL, attribute names must typically match the program variable names. For instance, in the previous example, `IPDest` must be a variable defined in the monitored

program. This association between the attribute names and the program variables provide application-dependent monitoring which facilitate observing LSD systems based on the values of the variables used in these applications at run-time [32]. Both the primitive and composite event body must follow the EVENT keyword and proceed the *event name* in HESL. The value of the attribute can be a *number* (integer or float point) or a *string* designated by quotes.

The HESL provides ANY, a special reserved word that can be assigned as an attribute value to indicate any range of values*. This has a significant use when the event attribute is only important in the event correlation expression and a specific value is irrelevant in the monitoring demand. In other words, this reserved word (ANY) is needed when a primitive event has an attribute whose value is interesting only when it is compared with an attribute of another primitive event. In the following, we present examples of defining events:

Event Example 1: Assume that one wants to define *warning* events (*AudioMixing*) that occur in function of the *AudioServer* program running in a specific machine called *dragon.* This event can be defined as follows:
EVENT= { ModuleName=AudioServer, FuncName=AudioMixing, Immediate;
EventType="Warning", Machine="dragon" } MixWarnings.

Event Example 2: If naming specific machine name is not important. Then the Machine attribute can be completely eliminated from the MixWarnings definition. However, assume that one is interested in discovering warning events that occur in both *AudioMixing* of *AudioServer* programs and *AudioReceive* of *RMPS* programs in the same "*machine*". In this case, naming a specific machine is not also important but it is important to be used in the correlation expression. For this reason, the MixWarnings is redefined as follows:
EVENT= { ModuleName=AudioServer, FuncName=AudioMixing, Immediate;
EventType="Warning", Machine="ANY" } MixWarnings.

EVENT= { ModuleName=RMPS, FuncName=AudioReceive, Immediate;
EventType="Warning", Machine="ANY" } RMPSWarnings.

*ANY is used for numbers and "ANY" is used for strings attributes.

TABLE 3.3

BNF OF THE ENVIRONMENT SPECIFICATION LANGUAGE

<EnvSpecs> ::= <Domain>; <EnvSpecs> | <Domain>& <ModuleSpecs>
| <Domain> @ <SuperDomainInfo> & <ModuleSpecs>

<Domain> ::= <DomainName> = <MachineList>
<SuperDomainInfo> ::= <SuperDomain> ; <SuperDomainInfo> | <SuperDomain>
<SuperDomain> ::= <SuperDomainName> = <DomainList>
<DomainList> ::= <DomainName>, <DomainList> | <DomainName>
| <SuperDomainName>, <DomainList> | <SuperDomainName>

<ModuleSpecs> ::= <Module>; <ModuleSpecs> | <Module>;
<Module> ::= <ModuleName> = <ModuleInfo>
| <ModuleName> = *
<ModuleInfo> ::= <ModuleLoc>, <ModuleInfo> | <ModuleLoc>
<ModuleLoc> ::= <DomainName> | <MachineName>
| <DomainName> − {<MachineList>}
<MachineList> ::= <MachineName>, <MachineList> | <MachineName>

Notice that `Machine` attribute has the value `ANY` in both events because the value of this attribute is not important in the event definition level. This implies that `Machine` attribute in this example can not be used to filter out events before checking the correlation expression that constitutes both events. In the Section 3.3.5, we show how an example of filter program that correlates these two events based on `Machine` attribute.

3.3.4 Environment Specifications

Large-scale distributed systems may involve any number of machines, sites, domains and networks. For example a distributed interactive simulation (DIS) application may support hundreds of nodes and tens of inter-networked sites. Managing and distributing agents

in such environment is a challenging task by itself. In order to facilitate and automate the task of agent organization and allocation, MSL provides the *Environment Specification Language (ESL)*. Consumers may use ESL to describe the run-time geographical distribution of the application entities/process.

Table 3.3 presents the formal specification of ESL in BNF. ESL provides a declarative and comprehensible interface to describe the run-time environment of an application. ESL has three major parts: machine distribution, domain hierarchy and event producers also referred to as application distribution. In the first part, consumers divide the machines that the application entities occupy during execution into distinct domains such that no machine can exist in more than one domain. Usually, domains are identified based on geographical distances and node clustering. For example, a LAN of workstations in one site can be considered a domain. However, consumers are free to choose *logical* domains based on other criteria. For example, machines can be classified based on configuration and CPU power, or based on the running application. The latter is useful for asymmetric distributed systems such as DIS applications where nodes are divided into groups and each group simulates a different entity (aircrafts, tanks, soldiers, ..etc). In this case, it may be more useful to classify nodes based on their functionality. Once all machines are contained in exclusive domains, the consumer may describe, if one exists, the hierarchical relation between these domains by including previously specified domains into *superdomains*. Again, no domain should exist in more than one of the superdomains. A superdomain may contain some of the previously specified superdomains to build the hierarchy to the root. For each level (l) in the hierarchy, the set of superdomains in l must be complete (i.e., contain all domains or superdomains in one level below l) and distinct (i.e., no domain or superdomain could exist in two different superdomains). A superdomain can not contain itself. If no superdomain is specified, the system will create a virtual superdomain that includes all listed domains.

The third part, application distribution, describes where the application processes are running in term of machines, domains or superdomains. ESL has special reserved characters such as $*$ and $-$ to provide a very flexible and easy to use language. The special character $*$ (all) indicates that the process/module runs in every defined machine. The special characters $-$ (except) excludes machines from domains. Table 3.4 presents a simple example of ESL. It shows that *ODU* and *VB* are domains (could be physical or

TABLE 3.4

ENVIRONMENT SPECIFICATION EXAMPLE

```
ODU = dragon, unicorn, orca;
VB = cyclops, harpy, ogre;
VA_State = ODU, VB

RMPS = *;
XTV = ODU, VB − { cyclops };
Sess = dragon, ogre;
```

virtual domains) that have the corresponding machines: "dragon", "unicorn" and "orca" in ODU, and "cyclops", "harpy" and "ogre" in VB. The domains ODU and VB are contained in one superdomain called *VA_State*. In addition, ESL in Table 3.4 states that the target processes of the application are `RMPS`, `Sess` and `XTV`, such that the first one is located *every where*, the second one is located in ODU and VB except cyclops, and the third one is located in dragon, cyclops and griffon machines.

3.3.5 Filter Specifications

After the application environment and associated events have been specified, the monitoring agents are configured and prepared to receive and process consumer monitoring requests or *filter programs*. Section 4.3 will discuss the process of configuring and organizing the monitoring agents. However, in this section, we focus on the *High-level Filter Specification Language (HFSL)*, which is used to subscribe new monitoring requests and to delete and modify former demands. The semantic model of HFSL was described previously in Section 3.1. Table 3.1 shows the HFSL syntax in BNF.

A filter consists of three main parts: the *event expression (EX)*, *filter expression (FX)*, and action. Each filter definition/program has `FILTER` as prefix, and the filter name as a postfix. Consumers define a filter program to describe an interesting event correlation [47] or pattern to be detected. The event correlation may specify a relation between

events occurrences, in general, and/or the relation between the attributes of different events. The event expression enables the consumers to specify the relation between the occurrences of abstracted events such as both events `MixWarning` and event `RMPSWarnings` have occurred: `MixWarnings` $\wedge$ `RMPSWarnings`. Consumers can also narrow down the relation of the event correlation to the attributes level using the filter expression in HFSL. The filter expression is composed of a set of predicates where each predicate represent two attributes of events (different or same events) compared by a logical relation such as $<, >$ and $=$. Events attributes are designated by the event name as a prefix in the filter expression (e.g., MixWarnings.Machine means the machine name attribute in MixWarnings event). Predicates are linked via logical operators called filter operators `Filter_Op`. Similarly, events in the event expression are linked together using event operators `Event_Op`. To avoid an ambiguous expression, parentheses are used to determine the precedence of the filter and event operators. Event and filter operators are kept independent even though they are identical in the current implementation, because event relation could comprise other operators that are not applicable in the filter expression, such as `After` and `Before` event operators. Therefore, event expression and filter expression constitute different levels of event correlation granularity which is important for increasing the expressive power of the monitoring language. It is important to notice that an event attribute in a filter expression may be assigned different value from the event definition. In this case, the effective attribute value is that of the filter expression. We call this feature *event attribute overloading* and its advantage is discussed in Section 3.3.7. If an event attribute is assigned a value in the filter expression, then this value is used instead of the default value in the event definition.

As the monitoring agents receive specified events, they work collaboratively to evaluate the event and filter expression as will be described in Section 4.2. As a result of agent collaboration, if both event and filter expression are evaluated to *true*, then the list of actions described in the `Action` part is performed. The action specification is the topic Section 3.3.6.

Filter Example 1: Using the HESL definitions of `MixWarning` and `RMPSWarning` events, we can specify an event correlation between these events such that they both generated in the same machines name as follows:

FILTER= [$(MixWarning \wedge RMPSWarning)$];
[$(MixWarning.Machine{=}RMPSWarning.Machine)$];
[FORWARD];`Warnings_Correlation_Filter`.

Since this filter example represent the occurrence of both events in the same machine, the machine name is only inspected only during evaluating the filter expression. This is why `ANY` is correct to assign as a value to `Machine` in Section 3.3.3.

Filter Example 2: The event expression (EX) of the filter program basically represents a composite event since it consists of expression of primitive and composite events. If the relation between the attributes of these events is not an issue, then filter expression can be set to `TRUE` to indicate this fact. In this case, the filter program represent a composite event as shown in this example:
FILTER= [$(MixWarning \wedge RMPSWarning)$];
[$TRUE$];
[FORWARD];`Warnings_Filter`.

This filter is to detect the composite event that represents the occurrence of MixWarning and RMPSWarning without any conditions. In Section 4.3, the method by which the filter specification gets fragmented and distributed among the monitoring agents (LMA and DMA) in the monitoring hierarchy is discussed.

3.3.6 Action Specifications

Informally, *actions* describe what will be done when the desired event pattern (correlation or composition) is detected. The filter program may include one or more actions that are performed in sequence. Table 3.5 presents the formal definition of the *High-level Action Specification Language (HASL).* There are five kinds of actions defined in HASL. These are described as follows:

Executing Programs: An action can be the executing of a program designated by "program name" and "path name", and accessible to the monitoring agents in the application environment. An absolute path name must be given, otherwise a path name relative to

TABLE 3.5

BNF OF THE HIGH-LEVEL ACTION SPECIFICATION LANGUAGE

<Action> ::= <Exec> | <Event_Name> | <Filter_Rinc> |
<Filter_Register> | **FORWARD**
<Exec> ::= <Path Name> <Program Name>
<Path Name> ::= <String> / <Path Name> | <String> /
<Program Name> ::= <String>
<Filter_Register> ::= <Identifier> = <Att_Name>
<Filter_Reinc> ::= **ADD** <Filter>; <Filter_Reinc> |
DEL <Filter>; <Filter_Reinc> |
MOD <New Filter>; <Filter_Reinc> |
ADD <Filter>; | **DEL** <Filter>; | **MOD** <Filter>;
<New Filter> ::= <Filter>.**EX**= <Event_Expr> |
<Filter>.**FX**= <Filter_Expr> |
<Filter>.**EX**= <Event_Expr>; <Filter>.**FX**= <Filter_Expr>

---

the LMA directory (also the application directory) is assumed.

Sending an Event: An action can be generating a new event by the monitoring agent that detects a filter correlation. As discussed in Section 3.1, this event could be used by the monitoring agents to summarize number of detected events or to trigger some other filters. This is significant for suppressing the events forwarded by LMAs and, thereby improving the DMA performance and minimizing the network overhead (intrusiveness).

Filter Incarnation: Filter incarnation is the process of adding a new filter, detecting or modifying an existing filter automatically at run-time. This feature has a significant impact on the dynamism and expressiveness of the monitoring system. This feature is discussed in detail in 3.1. Adding new filters means activating pre-defined filters that have not been submitted to the system. This is specified using a special reserved word (`ADD`) with the pre-defined filter name. On the other hand, deleting or modifying must be

performed on an existing filter that consumers subscribed to. This is specified using the reserved words, MOD and DEL, with an active filter name. When modifying an active filter, consumers must specify which parts to modify: event expression (EX), filter expression (FX), or both. This can be designated by appending the filter name as a prefix to EX and/or FX. The resulting EX and/or FX are the effective filter parts after the subscription is completed. Adding, deleting or modifying a filter dynamically at run-time may create inconsistency in the agent's state until the subscription process is completed. Section 4.3.4 presents an algorithm to resolve this problem.

Filter Registers: Consumers can create a set of virtual registers called filter registers by defining variables in the filter action part. These registers are used to restore attribute values of received events. This can be simply specified by the consumer by assigning the attribute value of an event used in EX or FX to a filter register. Filter registers are effective when used as an attribute in the filter incarnation. Section 7.2.2 shows an example of a filter program that uses filter registers and filter incarnation.

Forwarding Monitoring Information: This is the simplest action type. Consumers may request either to forward a notification of detecting an event correlation in a filter or to forward the event information itself. The former case is used consumers are mainly interested in the event correlation itself such as detecting a bug or error in the program. However, the latter case is used when information of the primitive events is requested such as generating customized trace history of the program execution. In this case, a consumer may desire to log selected events which are detected by the monitoring agents. Examples for both cases are discussed in detail in Chapter 7.

3.3.7 Language Design Features

Although the focus of this thesis is not on language design, MSL represents one of the major contribution of this work. MSL provides new characteristics that support the *expressiveness* and *usability* criteria discussed in Section 3.3.1. In the following, MSL features are discussed.

Expressiveness

The expressive power of a monitoring language is measured by the ability for describing a specific and general monitoring application [3]. As will be described in Chapter 7, MSL is used to define many monitoring applications for debugging, application steering and fault recovery. The monitoring system is used for developing a number of real case examples and applications in an IRI distributed system environment. As a result of experiments conducted in the IRI system, MSL proves its ability to specify and control classical problems in distributed system management, such as customized traces and slow clients, which will be discussed in detail in Chapter 7. Although the monitoring architecture is geared toward certain applications (reactive control and debugging), MSL can be utilized in other monitoring applications, such as performance measurements, security, and testing and verification of distributed systems. The MSL expressiveness stems from the following design features:

- *Imperative Power:* Although MSL is classified as a declarative language, the HASL enables a consumer to utilize the imperative power supported by filter incarnation and new event generation. Consumers can define recursive or looping constructors in a filter using such features. The `if-else` programming language constructor is emulated in MSL using a cascading set of filters.

- *Multi-level Abstraction:* The different level of event abstraction, such as primitive events, composite event (EX) and event correlation (EX and FX), is a key advantage in the providing various expressive power without sacrificing simplicity (ease of use).

- *Regular Expression:* The event and the filter expression inherits the expressive power of the regular expression. Regular expressions are widely used for specifying event sequences because of their expressiveness and their implementation simplicity [29].

- *Event Reuse:* Composite events inherit the attributes of all contained events (primitive or composite events). Thus, in the filter expression, composite events can be associated with attributes of the containing events without referring to the original event name. This reuse of event attributes enables using composite events transparently thereby contributing to the expressiveness and the simplicity of the MSL.

- *Event Attributes Overloading:* Event attributes take a default value specified in the event definition. However, in the filter expression, an attribute value can be overloaded (i.e., changed) with another new value. This new value will be effective in the DMA and the LMA retains the original default value.

- *Filter Overloading:* Modifying a filter (EX or FX) is a form of overloading since it is like replacing the current filter with a new one. This operation has a significant impact on the dynamism of the filter programming and, thus, the expressive power of the MSL.

- *Supporting Special Keywords/Operators:* MSL supports a number of reserved words such as `ANY`, `ALL`, `FORWARD` that simplify certain consumer tasks and improve the expressiveness of the monitoring language. Some of these keywords are discussed previously in this Section.

Ease of Use

The second criteria used to evaluate alternative approaches of monitoring languages is ease of use or simplicity . Programming languages, in general, gain simplicity mainly from two facts: (1) *simple syntax and data structure* and (2) *simple semantics.*

Simple Syntax and Data Structure: The simplicity of the MSL syntax stems from the following facts:

- *Easy to learn:* Learning MSL entirely consists of mastering about 10 key words and simple constructs.

- *Supporting Program-dependent Monitoring:* As described in Section 3.3.3, consumers can conduct program-dependent monitoring, if the variables names defined in the program are utilized as event attributes in the HESL and HFSL. MSL automatically creates a mapping between program variables and event attributes. Without this feature, consumers have to define this mapping explicitly.

- *Simple Event Schema:* Consumers may specify any arbitrary event format in a declarative way by simply making a list of event attributes. In addition, the event specification (HESL) enables a consumer to assign values to the event attributes and

does not require specifying attribute type (e.g., `int, float or string`). This is unlike the CORBA IDL [67] abstraction which requires each attribute type to be determined and does not permit the user to specify values in the event constructor.

- *Simple Constructs:* MSL uses intuitive and simple constructs that are commonly used in any programming language (e.g., using regular expressions and logical operators). In addition , MSL provides various special characters such as $*$ and $-$ to simplify the filter programming task.

- *Supporting Error Checking:* Writing a program using any language is always an error-prone process. Thus, MSL supports syntax and semantic error checking and alerts the user, if necessary. Parentheses missing in an expression or specifying a process within an unrecognized domain or machine are examples syntax error. On the other hand, a domain containing itself, or a domain that forms a cycle in the hierarchy tree (see Section 3.3) are examples of semantic checking.

Simple Semantic

The program semantic is the meaning of the program [70]. Many of the issues that support the expressiveness of MSL also support the simplicity of the MSL semantic including:

- *Declarative:* MSL components, HESL, HFSL and HASL, provide a simple and declarative semantic to define monitoring requests. For example, in HFSL, consumers define their monitoring demands by specifying the event correlation expression (EX and FX) and the action to be performed. No programming effort or complex constructors are required to define a monitoring demand. If consumers desire to filter events in the LMA level, then they simply have to specify the attribute values in the event format (HESL).

- *Basic Expression Operators:* MSL supports basic and simple operators (AND, OR and NOT). However, as shown in [29], advanced operators can be constructed based on such basic operators. Therefore, using basic operators do not limit the expressive power and also provides a simple interface and efficient processing language [28].

- *Expression Operators Precedence:* To provide an unambiguous expression for composing or correlating events, closed parentheses, such as `()`, are used to determine the operators precedence in expression evaluation.

Efficient Processing

This characteristic is inherited from the simplicity and the expressiveness of the MSL itself. Simple syntax and semantic enable a faster scanning, parsing and processing of the MSL components. Moreover, MSL is a compiler-based (as opposed to interpreter-based) language. Therefore, this facilitates run-time optimization and dynamic linking thereby improving the execution performance and optimizing memory usage. This comparison and contrast of compiler-based and interpreter-based languages is discussed in Section 3.3.2.

3.4 Summary

This chapter describes the monitoring model that includes the event producers, event consumers and monitoring operations abstractions. It then discusses the monitoring language specifications which define the user/system interfaces based on these abstractions. The program run-time behavior is abstracted as events notifications emitted from the program during its execution and defined using High-level Event Specification Language (HESL) shown in Table 3.2. Users (or event consumers) monitoring demands are abstracted as "filter" programs that detects only interesting events and suppresses other events. A filter program describes the events of interest, the relation between these events (event expression or EX) and the relation between the events attributes (filter expression or FX). Applications events are continuesly monitored and if the event correlation described in EX and FX is detected, then the "action" specified in the filter program is performed. The monitoring model supports *filter incarnation* feature that enables users to activate, deactivate or modify the monitoring demands (filters) automatically at run-time based on detected events. The filter programs and actions are defined using High-level Filter Specification Language or HFSL (Table 3.1) and High-level Action Specification Language or HASL (Table 3.5), respectively. The monitoring language also includes Environment Specifications Language (ESL) (Table 3.3) which consumers use to describe the geographical distribution of the application entities (i.e., processes) to be monitored. While the monitoring model improves the monitoring dynamism via event-subscription-based abstraction and filter incarnation, the monitoring language also uses declarative high-level language to support simple and flexible interfaces.

CHAPTER IV

SYSTEM ARCHITECTURE

This chapter presents alternative filtering techniques used in distributed event management. It then describes algorithms and protocols that support the hierarchical filtering-based monitoring approach. This chapter provides the reader with a tour of the monitoring system, from user specification and processing to actions execution. Techniques used to implement the architecture such as automatic agent allocation, event and filter decomposition and distribution, dynamic subscription, and event detection are described. In this chapter, we also state the impact of these techniques on achieving the dissertation objectives.

4.1 Alternative Filtering Architectures

The event filtering architecture has a major impact on the scalability, performance and perturbation effect of monitoring large-scale distributed systems. In the following, the existing event filtering mechanisms are classified according to their filtering architecture. Then each filtering architecture model is evaluated based on the requirements of monitoring large-scale distributed systems. There are several models of event filtering architectures as described below:

Decentralized Event Filtering Architecture: In a this architecture, all generated events are sent to all consumers. Each consumer performs the filtering operation based on its own interest (see Figure 4.1-a). This filter architecture is used in some existing monitoring systems, such as filters in communication protocols [12, 59, 63, 94], system and network management [16, 83] and distributed systems [14]. However, it is not sufficient for monitoring large-scale distributed systems for the following reasons:

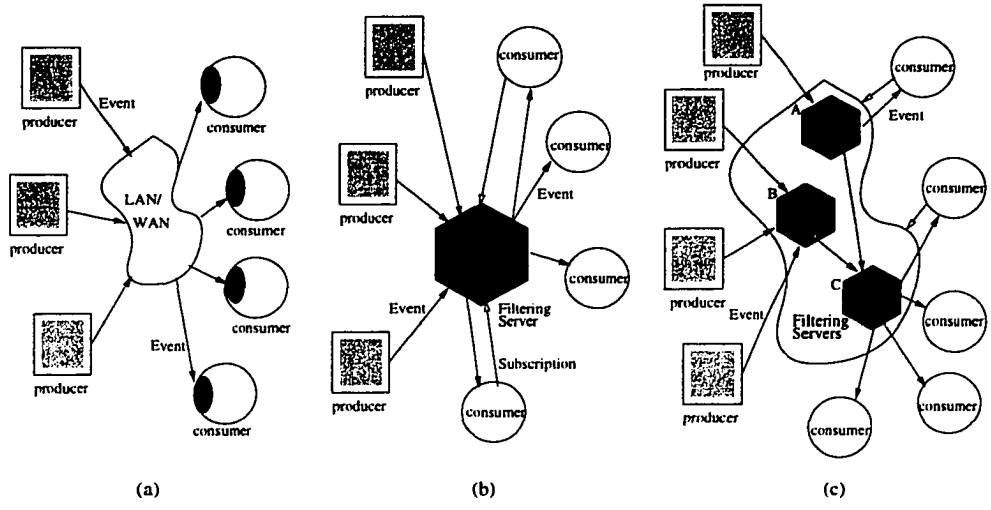


Fig. 4.1. Event Filtering Architectures. (a) Decentralized Filtering Architecture. (b) Centralized Filtering Architecture. (c) Semi-Distributed Filtering Architecture.

- The architecture imposes a processing overhead on consumers since they are exposed to every generated event in the system. Consequently, severe degradation in the performance of the event filtering mechanism may occur when the number of producers (or events) is increased.
- The architecture is not appropriate to be used in WAN environment (such as the Internet) because of the drawback of broadcasting a large amount of events to all consumers in the network. This may cause packet flooding and congestion problems in the network. Therefore, this architecture can not efficiently accommodate consumers and producers in a WAN environment.
- It requires the availability of a high-speed network to deliver a large volume of events from producers to consumer hosts.
- Because of the redundant effort performed by consumers in this architecture, computational and buffering resources are not utilized efficiently.
- This architecture could be appropriate if consumers subscribe to most of the events

in the system. However, this is not usually the case when monitoring large-scale distributed systems.

Centralized Event Filtering Architecture: In this architecture, event filtering is performed at a central server called the event filtering server, which is located between the producers and the consumers (see Figure 4.1-b). This architecture is typically used when network and consumer hosts are the processing bottleneck, rather than the event filtering server. In this case, a centralized filtering server helps to offload work from the network and the consumer hosts. The filtering mechanism in active database systems [27, 29, 93] represent a centralized architecture since events are generated and filtered in the same machine. Examples of monitoring systems using this architecture are presented in Section 8.1. The centralized filtering architecture is inefficient for monitoring large-scale distributed systems for the following reasons:

- In general, a centralized approach does not scale well in large-scale distributed environment where a large number of consumers and producers may exist.
- This architecture may be appropriate if consumers and producers reside on the same host or are separated by a small geographical distance (such as in a LAN environment). However, this does not fit the requirements of monitoring large-scale distributed systems described in Section 1.2.
- It confronts the traditional problems of centralized systems: performance bottlenecks and a single-point of failure of event filtering server.

Semi-Distributed Event Filtering Architecture: Some existing monitoring systems use a limited distributed filtering approach. In this architecture, the process of event filtering may span more than one filtering servers, many *local* and one *central* filtering servers [69], located between the producers and consumers (see Figure 4.1-c). Examples of monitoring systems using this architecture are presented in Section 8.1. Although this architecture seems to be more scalable and efficient than the previous ones, it is still suffers the following limitations:

- It offers a limited scalability with the increase of producers, consumers and events because of the central filtering server. Replicating the central server does not solve

the problem because the distributed management and coordination mechanisms between these servers are still missing in this architecture. This is despite the fact that replication in some of these system is also not feasible such as [69].

- This architecture also suffers from a single-point of failure since the central server may fail causing the entire system to fail.

- This architecture enables event filtering to take a place at different locations simultaneously which may reduce the amount of events flow in the network. However, this is still insufficient for monitoring large-scale distributed systems, such as interactive distance learning or distributed interactive applications, because it requires all events detected by the local servers to be forwarded to the central server and some of these events are not desired. This implies that interconnected nodes in different LANs have to forward events across a WAN, such as Internet or Intranet, in order to obtain global monitoring. This forwarding is unnecessary and it may cause sever network problems and increase monitoring intrusiveness.

For these reasons, we call this architecture that offers limited distributed advantages a *semi-distributed* architecture. Therefore, this type of architecture can be classified as a decentralized or centralized approach based on the emphasis location of the filtering mechanism. If most of the filtering is performed in the producer or consumers edges, then it is considered a case of decentralized architecture. On the other hand, if the filtering is mostly performed in the central node, then it is obviously a type of centralized architecture. An alternative technique is needed to intelligently limit event flow and fully utilize the distributed processing feature of the WAN, such as embodied by the Internet. In the following section, a distributed hierarchical filtering architecture is presented and then evaluated to illustrate the improvement over the above described architectures.

4.2 Hierarchical Filtering-based Monitoring

In our monitoring architecture, the task of detecting primitive and composite events is distributed among dedicated monitoring programs called *monitoring agents* (MA). MA is an application-level monitoring program that runs independently of other applications in the system and communicates with the outside world (including producers and consumers)

via message-passing. For example, HiFi has two types of MAs: *local monitoring agents (LMA)*, and *domain monitoring agents (DMA)* (see Figure 4.2). The former is responsible of detecting primitive events generated by local applications in the same machine while the latter is responsible of detecting composite events which are beyond the LMA scope of knowledge. One or more producer entities (i.e., processes) are connected to a local LMA in the same machine. Every group of LMAs related to one domain (geographical or logical domain) is attached to one or more DMAs*. These DMAs are also connected to higher DMAs to form a hierarchical structure for exchanging the monitoring information. In this section, the architecture, agent organization, communication and management protocols of HiFi distributed hierarchical monitoring are described.

4.2.1 Distributed Filtering Management Protocol

Event filtering is a key component in the monitoring architecture. In HiFi, the event filtering is divided into three different levels:

Identity-based Filtering

Primitive events sent from the producer are actually generated by the *Event Reporting Stub (ERS)*, which is library linked with producer code during compilation. ERS is a HiFi supported library provided to facilitate the code instrumentation process as discussed in Chapter 5. ERS generates only the interesting events involved in at least one filter subscription. ERS checks the events identity (i.e., name) and suppresses all events reported by the producers that are not contained in an active filter. This "event-identity filtering" represents the first filtering level in the hierarchy; this implementation is described in detail in the Chapter 5. ERS forwards interesting primitive events to its local LMA in the same machine via a UNIX communication channel [86] as described below. Because of the processing and communication overhead ERS incurs by this level of filtering, users have the option to enable or disable this level of filtering. In this case, ERS delegates this task to the LMA by forwarding all reported events as will be discussed in more details in Section 5.1.3.

*More than one DMA may be used for fault tolerance

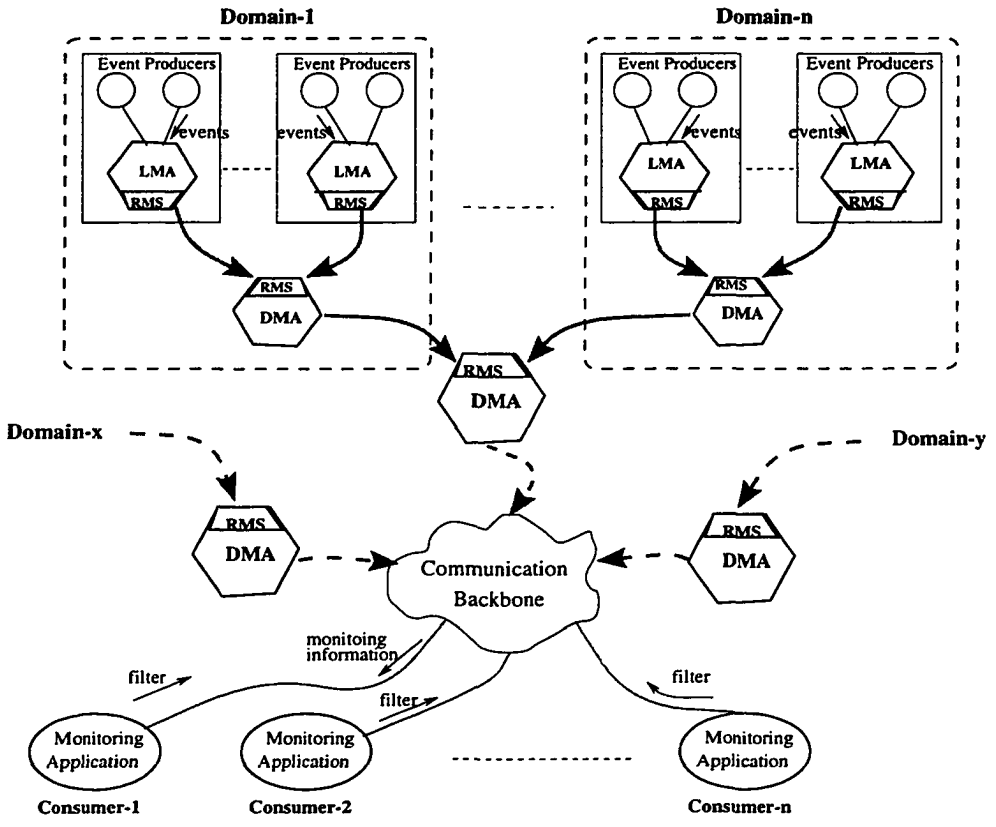


Fig. 4.2. Hierarchical Filtering-based Monitoring Architecture.

Content-based Filtering

Content-based filtering is often also referred to as "local filtering". When a primitive event is received by an LMA, the LMA checks the filter internal representation which is a Directed Acyclic Graph or DAG that holds the local filtering information for any matching event. If this match is found, then the contents (attributes) of the received primitive event is checked according to the event subscription information. For example, ERS may generate the event `WarningEvent` as it is subscribed to by the use. However, the LMA may *reject* this event because, for instance, the *ModuleName* (see Table 3.2) of this event does not match the user event/filter specification. So, at this level (LMA) events are filtered based on the contents of the events or the values of the event attributes.

If an primitive event of interest is detected, LMA notifies its containing DMA and/or a group of DMAs in the next higher level of the hierarchy. This implies that the forwarded event must be part of another composite event or a filter for which a consumer has subscribed. LMAs may directly forward the detected primitive event to the corresponding manager(s), if this event is not contained in any correlation, and the consumers request *Forward* action (see Table 3.1).

Correlation-based Filtering

Correlation-based filtering is also called "domain filtering". The DMAs collect information from their local LMAs and check for any match of a composite event in their domains. A composite event in this sense means also any predicate of an event correlation expression that contains attributes of two distinct events, such as $SendEvent.size = ReceiveEvent.size$ where SendEvent and ReceiveEvent are different events. The EX or FX evaluation of a filter could be distributed among several DMAs based on the location of related events in each predicate. The process of decomposing EX and FX is explained in detail in Section 4.3.

Upon receiving a primitive event from LMA, DMA evaluates each related event correlation expression, typically *Event_Expr (EX)* and *Filter_Expr (FX)* (see Table 3.1), in its filter internal representation or PetriNets (PN) (see Chapter 2). If an expression evaluates to *true*, then either this expression represents the entire EX and FX, or it is a segment of the EX and FX. In the former case, the DMA declares detecting a user filter by notifying the corresponding consumers, and performing the action specified in the filter. In the later case, the DMA forwards this composite event (evaluation result) to the next higher DMA in the hierarchy which, in turn, checks for the existence of this composite event in any of its event correlation expressions (EX and FX). If this match is found, the DMA uses this composite event for further filtering and evaluation of EX and FX. Otherwise, this hierarchical communication continues until the composite event reaches a DMA that needs this information to complete evaluating EX and FX. It should be noted that if the first DMA detects a composite event, then there must be a higher DMA in the hierarchy that can utilize this composite event for further filtering and evaluation of EX and FX. On the other hand, if the primitive event is not detected by the first DMA (i.e, correlation expressions evaluates to *false*), then the DMA also forwards this event to its

containing DMA which may have more global knowledge about the events occurring in the system. This hierarchical filtering continues until the primitive event is detected by a DMA and the process proceeds as described above or it gets rejected by the DMA root.

The number of levels in the monitoring hierarchy is dictated by the requirements of LSD systems. For instance, LSD systems that are delay-sensitive should reduce the number of levels to minimize the communication latency. On the other hand, for LSD systems producing a very high-volume of events and the MAs not residing in powerful machines, deeper hierarchy can help in distributing the monitoring load and alleviate any performance bottleneck in the monitoring architecture. However, in most cases, two to three hierarchical levels are sufficient for monitoring LSD systems. As discussed in Section 3.3, ESL of the monitoring language has a significant role in controlling the depth (number of levels) and breadth (number of LMAs) in the agents hierarchy tree. It is also important to mention that even if received primitive or composite events have a match in the DMA PN, this does not necessarily imply evaluating the EX or FX in this DMA because other events composing this EX or FX may not exist yet. In this case, the DMA buffers the received event until all events information composing EX or FX become available. Then, DMA restores the event information and evaluates the event or filter expression accordingly.

4.2.2 Dynamic Agents Hierarchy

Prior to any monitoring operation, the consumer must describe the physical or geographical distribution of the application that he/she intends to monitor using the declarative and comprehensible interface called *Environment Specification Language (ESL)* described in Section 3.3. HiFi (subscription component) parses and validates the ESL script, then uses the information provided in ESL to construct the LMA(s) and DMA(s) hierarchy as described later in Section 4.3.3. Once the agent hierarchy is established, communication and monitoring can take place. However, this hierarchy is not static and may change dynamically according to agent processing load in the monitoring application. In this section, we discuss the techniques used to build dynamic and adaptable monitoring agents.

Adaptable Monitoring Agents

One of the challenging issues in monitoring large-scale distributed systems is the large number of generated events that could swamp the monitoring agents. A dynamic re-configuration technique is required to achieve a higher performance and reliability in the monitoring process. Although HiFi uses several filtering optimization techniques (discussed in Section 5.3.4) to achieve high-performance monitoring, dynamic re-configuration is still needed to adapt to the increasing monitoring load.

LMA Load Adaptation: LMAs can dynamically increase the default queue length associated with each producers if the threshold is reached. In addition, consumers can assign an LMA for all processes in the same machine, or an LMA for each process. This decision is made during the instrumentation process and it provides flexibility to users to specify the LMA architecture based on the application performance requirements.

DMA Load Adaptation: As several LMAs may report to a single DMA, the DMA has more potential to be a performance bottleneck. A DMA creates and monitors an event queue for each LMA in the same domain. If the event queue length of a DMA exceeds the maximum threshold specified, the DMA realizes that the total event forwarding rate (λ) is higher than the event processing rate by the DMA (μ). Consequently, the DMA sends a request message to an associated LMA in a different machine to create (`fork`) another DMA. The LMA that has the minimum forwarding rate is selected for this purpose. Then, the original DMA partitions the domain by requesting some LMAs to relinquish its service (connection and forwarding) and get associated with the other DMA. In order to obtain an efficient distribution of the monitoring load, DMA partitions the LMAs in the domain based on ρ_i of each DMA queue (q_i) such that $\rho_i = \lambda_i/\mu_i$ for all q_i in the DMA. This means that both processing time (μ_i) and forwarding rate (λ_i) of each LMA are considered in this evaluation. This is important to consider the overhead of both events frequency and complexity in this process. The number of DMAs required can be determined by: $\lceil \rho_{DMA} \rceil$ such that $\rho_{DMA} = \sum_{i=1}^{n} \lambda_i/\mu_i$ (n total number of LMAs in the domain). After new DMAs are created, the original DMA sends the monitoring information in the PN to them, and a notification to the manager[†] to update the environment information. Then

[†]the manager here is the event consumer program that sent out the monitoring information.

the new DMA sends a request to its associated LMAs to resume forwarding events.

Agents Communication: Virtual Hierarchy

Three forms of agents communication are allowed:

- LMA-DMA which is used to forward detected primitive events from an LMA to a DMA,
- DMA-DMA which is used for forwarding composite events from one DMA to another in a higher domain in order to correlate events, and
- DMA-LMA which is for sending monitoring control information from a DMA to an LMA such as re-assigning an LMA to another DMA for load adaptation discussed in details in Section 4.2.2.

The monitoring agents (LMAs and DMAs) use Reliable Multicast Server (RMS) described in [4] for communication. LMAs and DMAs create and join *LMAGrp* and *DMAGrp* multicast groups, respectively (see Figure 4.3). This implies that events sent to DMAGrp, for example, are delivered to all DMAs in the network. However, monitoring agents utilize the *dynamic group masking* feature in RMS described in [4] to selectively send events to a subset of the entire group. In particular, the message that encapsulates a forwarded event must include the agents IDs (MachineName.DomainName) of intended recipients within this group. This enables RMS to dynamically filter out other members from this delivery. RMS also permits members outside the group to send messages to a multicast group reliably via *connect* request (`Connect` *GrpName*). For this reason, LMAs issue a *connect* request to RMS in order to establish a reliable multicast connection with DMAGrp. However, a DMA requests a connect to LMAGrp only when load adaptation is needed as described in the Section 4.2.2. DMA can directly multicast to one or more DMAs since they are all members in DMAGrp. Similarly, LMA can multicast to one or more DMAs after connecting to the DMAGrp. Although agents employ a hierarchical management protocol for event monitoring, it is a *virtual hierarchy* which is formed by LMAs and DMAs, over RMS reliable group communication.

Manager-Agent Communication

As described in the monitoring model in Section 3.1, the monitoring agents communicate with a number of managers representing the event consumers programs. This communication RMS is also used in this case to enable an efficient group communication between agents and managers in the same application. Managers usually send to agents event, filter and environment information which are necessary to perform the monitoring operations. On the other hand, agents forward event notifications or control information to one or more manager based on their subscription requests.

Using reliable multicasting, instead of TCP point-to-point, for agent communication has several key advantages in the architecture:

- It supports a dynamic and scalable dissemination of monitoring information to a group of consumers or manager programs simultaneously.
- It supports an efficient mechanisms for a multi-point communication between the agents. Multicasting eliminates the processing and network overhead caused by performing multiple sends for a single packet, thereby improving the system performance and minimizing its intrusiveness.
- Reliable multicasting significantly improves the robustness and survivability of the monitoring architecture in the presence of agents failures. Multicasting protects the system from network partitioning and agents isolation, where one or more agents malfunction and crash. In this case, other agents can communicate and negotiate recovery procedures. This is difficult to achieve in a point-to-point TCP connection without back up connections and ad hoc techniques, which, in turn, cause waste of resources and does not offer a scalable solution.

4.2.3 Advantages of Distributed Hierarchical Filtering

The distributed hierarchical filtering architecture has several advantages over centralized, decentralized or semi-distributed architectures.

- Distributed hierarchical filtering significantly improves the monitoring performance. This is due to monitoring load being widely distributed among the LMA and DMA

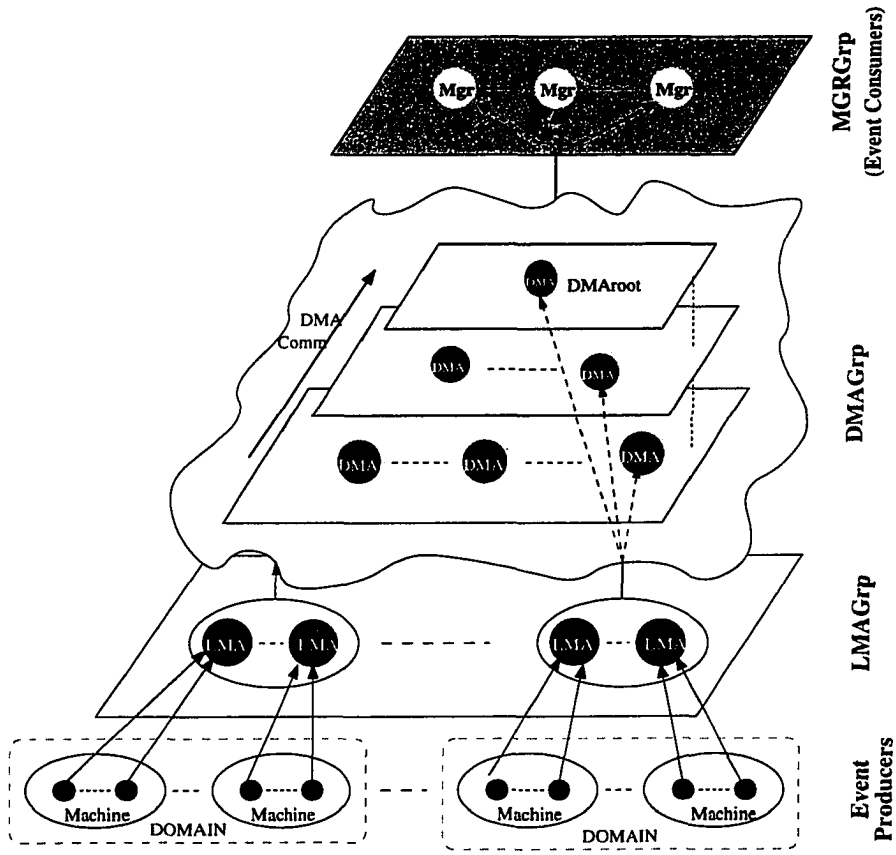


Fig. 4.3. Monitoring Agents Hierarchy.

groups which alleviate performance bottlenecks and increase the monitoring tasks concurrency.

- It scales well with the increase of producers, consumers and generated events. More LMAs and DMAs are automatically created to accommodate the increase in the monitoring load or the applications entities. As discussed in Chapter 5, users have full control in specifying number of LMAs per machine and number of application entities (processes) connected with each LMA.

- This architecture avoids a single-point of failure. In fact, failures some LMA and DMA agents can be recovered and the effect of single-point of failure does not in any case lead to a total termination of the monitoring operations.

- The amount of event flow is limited as events filtering and classification are localized in the area from which they originate or are generated. In this architecture, detected events are forwarded only to the concerned agents which must find these events interesting. This is unlike the semi-distributed architecture described in Section 4.1 in which detected events are forwarded to a central agent.

4.2.4 Hierarchical Monitoring Enhancements

In the hierarchical monitoring architecture described previously in Section 4.2, the primitive and composite events propagate up in the hierarchy until they are detected by the DMA or rejected by the DMA root. However, hierarchical communication can be significantly reduced if events are directly forwarded to the proper DMA(s) in the hierarchy that evaluate an EX or FX containing such events. This eliminates unnecessary processing and communication overhead, which improves the performance, and minimizes the intrusiveness of the monitoring system.

Definition: A *containing set* of an MA_i is a list of DMAs that includes the parent and the predecessor parents of this MA:

$$ContSet(MA_i) = \begin{cases} DMA_{root} & if\ Parent(MA_i) = DMA_{root} \\ Parent(MA_i) \bigcup ContSet(Parent(MA_i)) & Otherwise \end{cases}$$

A "short-cut" communication technique was developed as an optimization technique in the agent management protocol. This technique enables LMAs and DMAs to forward events directly to the concerned DMA(s). When an LMA detects a primitive event, it multicasts this event to DMAs within its containing set only. Each DMA, consequently, evaluates the correlation expression based on received primitive events and then similarly multicasts the resulting composite event information to the containing set DMAs which may need such composite event information for further filtering and correlation. The resulting composite event indicates that correlation subexpression is evaluated to *true* or *false*. In either case, forwarding the results is essential to complete evaluating of the EX

and FX expression by DMAs. Remember that forwarding composite events, in this case, occurs only if a DMA has a segment or subexpression of EX or FX. However, if a DMA contains the entire expression, then no further forwarding is necessary and the DMA performs the filter action if the EX and FX is satisfied. When EX and FX are fragmented and distributed into subexpressions, LMAs and DMAs are informed to which DMAs the subexpression evaluation results should be forwarded.

4.3 Monitoring Process

The monitoring operation comprises a number of stages including monitoring specification, program instrumentation, agent administration, consumer instructions, event detection, and information dissemination and presentation.

In this section, the overall view of the monitoring process is presented with a description of the protocols and algorithms used to execute the monitoring process. In Chapter 5, the component-level design and implementation used to accomplish these monitoring tasks will be presented.

4.3.1 Monitoring Specification

Prior to any monitoring operation, consumers must use the monitoring system languages (MSL) to describe the monitoring specification. Users start with specifying the environment specifications (ESL) and events specifications (HESL) that describe the application distribution and the interesting events. Then, users proceed to the next step which is application code instrumentation. This stage includes two basic steps: (1) inserting events sensors, called *user sensors*, in the program code in order to generate events from the monitored program, (2) running the application code through the *Instrumentation Preprocessor (IP)* which replaces the user sensors with extended sensors called *system sensors*, and (3) compiling and linking the instrumented code with an external instrumentation library called Event Reporting Stub (ERS). The implementation details of these steps are described in Chapter 5.

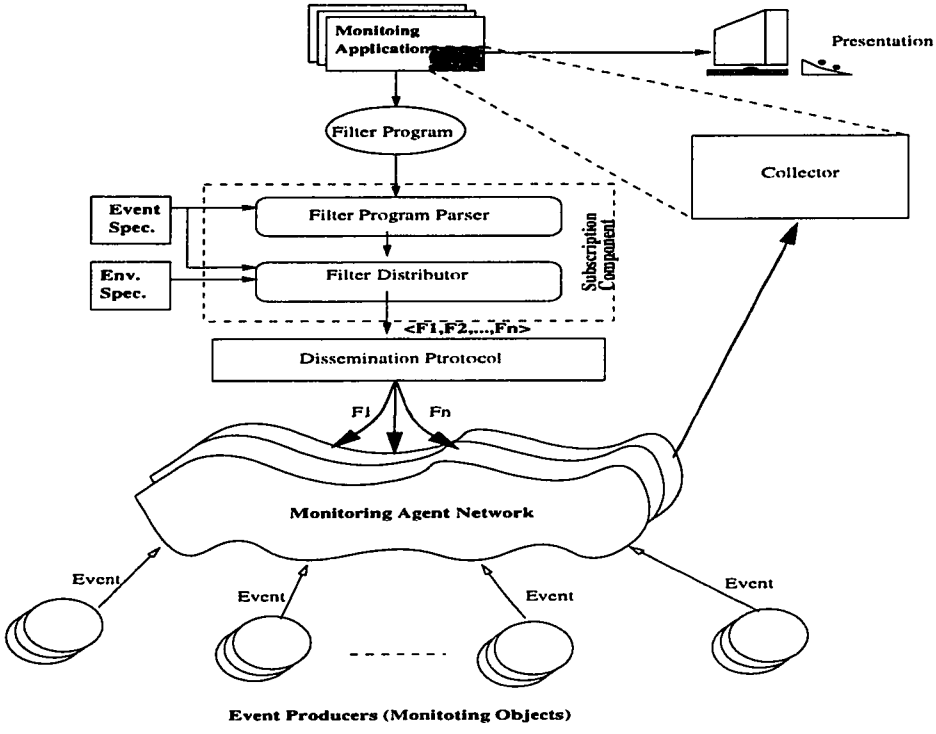


Fig. 4.4. Monitoring Process.

4.3.2 Monitoring-Knowledge Base

The monitoring system parses and analyzes the events and application environment specifications supplied by users as described in Section 3.3. This processing is not only important for validating the syntax but it is also essential for constructing the *monitoring-knowledge base* required for pursuing further monitoring operations. To perform these functions, the *Monitoring Language Processor (MLP)* program, which is part of the *Subscription Component*, is provided to consumers to use prior to the subscription process.

The monitoring-knowledge base is derived from MSL and contains the *setup information* which describes agents distribution, events and filters information. The implementation details of the structure and the acquisition of the monitoring-knowledge base are described in Chapter 5.

4.3.3 Automatic Agents Organization: Hierarchical Setup Protocol

After the application is instrumented, the program is ready for monitoring and consumers can now execute their programs and send their filters to monitor and control the running program. Consumers use the *Subscription Component* to process and send these filters to the monitoring agents. However, there are no agents or any monitoring entities running in the application environment, so far. Then, how LMAs and DMAs are created? Where are they located? And how do they recognize their roles and, thus, construct the agents hierarchy? The employment of a complex hierarchical agent structure that complies to a certain management protocol, may cause extra overhead in the agents administration. For example, operating the agents, distributing the roles, allocating tasks and synchronizing the communication between the agents are examples of administrative tasks which are performed before starting the monitoring operations. One approach is to delegate these tasks to consumers so that they can manually execute agent programs in the desired machines with the proper command-line arguments that define the role of each agent. However, these process is error-prone and far too complex for the users to handle.

The main objective of developing network and system management tools is to facilitate administrating systems and networks. It defeats the purpose if using management tools imposes a considerable overhead on the users. For these reasons, we developed a protocol that automates starting, allocating and setting up the agent hierarchy dynamically during the program execution and without the involvement of the users or consumers. Figure 4.5 shows the *protocol interaction diagram* between the parties of this protocol operation. In the following, we give a detailed description for the protocol.

1. *(Manager Program starts.)* The manager program is the first to start in the monitoring environment. When the manager program starts it performs the following:

   (a) Joins MgrGrp

   (b) Connects to LMAGrp

   (c) Waits for *all* LMAs to start and connect to MgrGrp.
   Notice, RMS sends a notification to the manager (MgrGrp) whenever a member joins or connects to the group.

2. *(Instrumented Program starts.)* When the instrumented program starts, it invokes

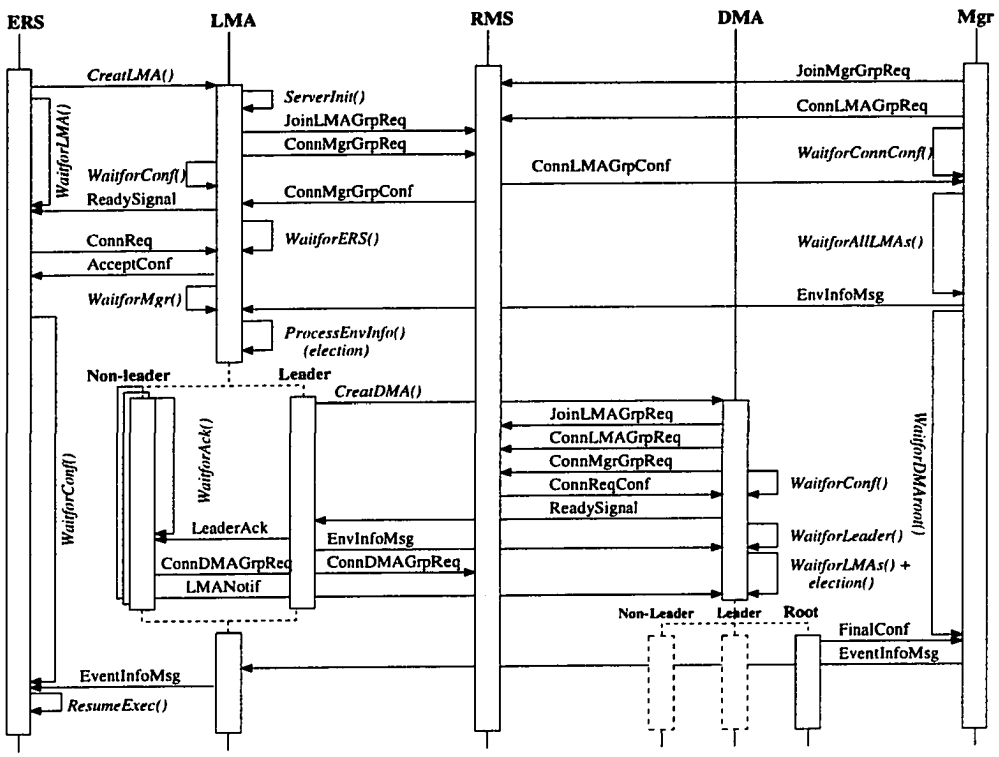


Fig. 4.5. Automatic Agents Organization Protocol.

the ERS for initialization using `ERSInit()`.

3. *(ERS starts.)* When `ERSInit` is invoked, ERS performs the following:

   (a) Initializes the UNIX sockets connection

   (b) Sets the signal handlers

   (c) Creates (`fork()`[‡]) an LMA

   (d) Then, ERS Waits for *Ready* signal from LMA

4. *(LMA starts and LMA-ERS connection established.)* After an LMA starts, it performs the following:

[‡]We assume that the agents binary exists in the same location or file server where the monitored programs exist.

(a) Joins/creates LMAGrp and then it connects to MgrGrp

(b) After the confirmation is received, LMA sends a *Ready* signal which is a SIGUSR UNIX signal to ERS to start the UNIX connection

(c) ERS connects to LMA which accepts and conforms this connection

(d) Then, LMA waits for the manager connection

(e) ERS waits for the final conformation of the hierarchy set up from LMA

5. The manager receives from RMS *connect notifications* of all LMAs. Notice that the managers know about the total number of LMAs from the environment specifications.

6. The manager multicasts the environment information (EnvInfo) to LMAGrp

7. *(LMA election process starts.)* Upon receiving EnvInfo from a manager, LMAs go through an election process based on the position of LMA name/ID in the EnvInfo table. In particular, the first LMA name in the LMAs list of each domain is the LMA leader. Users can also control this by arranging the machines in the ESL. In the example presented in Table 3.4, *LMA-dragon* and *LMA-cyclops* is the LMA leaders for ODU and VB, respectively. Therefore, LMAs are divided into two groups: a leader group that contains the LMA leader for each domain, and a non-leader group that contains the other LMAs. In the following, we describe what each group may do after the election completes:

   (a) (*LMA Leader.*)

      i. Creates the conducted DMA for this domain

      ii. After DMA starts, it sends a signal to its LMA creator

      iii. LMA Transfers the environment information to its DMA after receiving a *Ready Signal* from it.

      iv. Connects to the DMAGrp

      v. Sends an acknowledgment to non-leader LMAs to announce the DMAGrp (DMA) preparation to accept connections

   (b) (*LMA Non-Leader.*)

      i. A non-leader LMA waits for the LMA-leader acknowledgment in order to connect to the DMA of the domain.

ii. Once a non-leader LMA receives an acknowledgment from the LMA leader, it:

A. Transfers the environment information to its DMA after receiving a *Ready Signal* from this DMA.

B. Connects to the DMAGrp

C. Sends an explicit notification to the DMA

8. *(DMA starts and the election process.)* When a DMA is created by the LMA leader, it initializes the communication groups and sends a signal to the LMA leader for connection acceptance. Then, a DMA reforms the following actions:

   (a) DMA waits for the environment information to be sent by the LMA leader

   (b) DMA waits for connection confirmations from LMAs in the its domain

   (c) If all LMAs in the domain are connected to the DMA and EnvInfo is received, DMAs go through the same election process used by the LMA. As a result of this election process, DMAs are classified into: DMA leader, DMA non-leader and DMA root. The first two groups (DMA leader and non-leader) follow the same steps described for LMA leader and non-leader. This means the every DMA leader creates its containing DMA (superDMA) and this hierarchical construction continues until DMA root is reached.

   (d) (*LMA Root.*) If a DMA discovers from the EnvInfo that it is the DMA root, it immediately sends a final conformation to the manager confirming the completeness of the agents hierarchy.

9. The Manager then sends a multicast event information (EventInfoMsg) to the LMA-Grp

10. Each LMA consequently forwards EventInfoMsg to associated ERS(s) to resume execution. ERS uses the events information received to construct and send events.

11. ERS resumes the program execution and event reporting. The LMAs and DMAs are completely set up and ready for receiving monitoring tasks (filters).

This protocol is implemented and used in the HiFi start up process. The protocol also scales very well with the number of agents since DMAs at the same level and LMAs

operate concurrently and the effect of the hierarchy height is minimal. It is important to mention that process crashes or abnormal leaves from the multicast groups (ERS, LMAGrp, DMAGrp and MgrGrp) are immediately propagated to the rest of the agents which causes the agents to abandon this process and quit. This guarantees that the final confirmation is sent only if the entire agent hierarchy is constructed successfully.

4.3.4 Dynamic Subscription Algorithms and Protocols

After monitoring agents are distributed and organized (as described in Section 4.3.3), manager(s) receive a notification or a "ready signal" to start the subscription process. The subscription process consists of two steps: (1) event distribution, and (2) filter subscription. In this section, we describe algorithm and protocols developed for supporting both steps of the subscription process.

Events Distribution

After the events are specified by consumers using HESL, the MLP processes the event specification to construct a mapping between each "primitive event" and its corresponding "LMA". This PE-LMA mapping is important for LMAs to determine their event filtering role (i.e., which primitive event is interesting). This mapping is constructed based on the event and environment specifications. As shown in Figure 4.6, using HESL, an event can be mapped to a module name. Since there is many-to-many mapping between *Events* and *Modules*, a list of modules could be obtained. Then, using ESL, the module list can be mapped further to a list of domains, machines or both. The resulted list are then combined to obtain a list of LMAs from which this primitive event is originated. Notice that mapping from machine or domain name to LMA is straightforward since, in any domain, there is only one LMA per machine. This mapping is performed by *PrimEventToLMA()* algorithm shown in Figure 4.7. The PE-LMA mapping information is automatically disseminated by the manager to all LMAs after the hierarchical setup described in Section 4.3.3 is completed. Therefore, upon receiving PE-LMA mapping information, each LMA immediately determines associated events and configures its DAG accordingly.

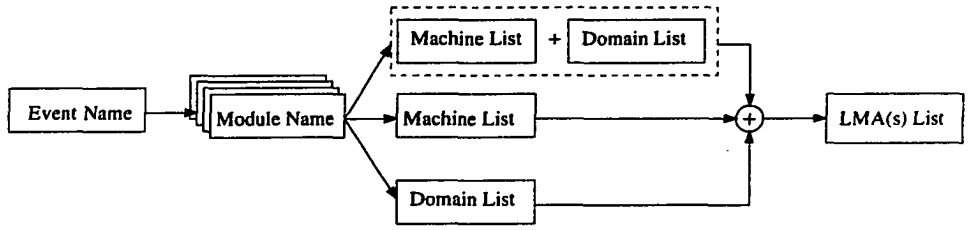


Fig. 4.6. Primitive Event to LMAs Mapping.

Filter Decomposition And Allocation

Figure 4.4 describes the steps of the subscription process, which starts by a consumer (manager) who issues a subscription or a *filter program (F)* through the MLP program. Then, the *filter parser* in MLP reads and parses the filter program to validate its syntax using the *event specifications*, which constitutes the valid formats of pre-defined events.

The MLP also constructs the monitoring-knowledge base of the parsed filters which include the filter decomposition and allocation information. This information is subsequently used by the *filter distributor* (1) to decompose the filter program (F) into subfilters (e.g., F_1,..,F_n) such that each subfilter represents a primitive event, and (2) to distribute the filtering responsibility (subfilters) among MAs. The *dissemination service* then uses RMS to multicast each subfilter component (F_1,.., F_n) to the assigned MAs. As a result, each decomposed subfilter is assigned to one or more LMAs based on the *environment specifications* which maps the events to their physical location. The filter distributor also determines the proper DMAs needed to evaluate the event expression (EX) and filter expression (FX), thereby detecting the event correlation specified in the original filter (F). The process of parsing, decomposing and allocating the filter program is called *filter processing* which is performed by the subscription component described in Section 5.2. In the following section, algorithms for *decomposing and allocating* the filter components (events, filter expression and event expression) are shown.

Input: Event as event name

Output: Machines that generate the input event (LMAList)

External Functions: *GetModules*(event): returns module names that produce *event*
GetDomains(mod): returns a list of domain names in which module *mod* executes
GetMachines(dom): returns machine names contained in the domain *dom*
IsMachine(mac): returns *true* if *mac* is a machine

```
PrimEventToLMA(Event)
  LMAList = Φ
  ModList = GetModules(Event)
  for j from 1 to |ModList| do
    DList = GetDomains(ModList[j])
    for i from 1 to |DList| do
      if (IsMachine(DList[i])) then
        LMAList = DList[i] ⋃ LMAList
      else
        LMAList = GetMachines(DList[i]) ⋃ LMAList
      end if
    end for
  end for
  return LMAList
```

Fig. 4.7. Event to LMA Mapping Algorithm.

Events Decomposition And Allocation Algorithms

The purpose of this algorithm is to extract all primitive events (subfilters) that constitute the event correlation (EX and FX) of a filter. Since all events in the filter expression must be contained in the event expression (see Section 3.3), considering EX alone in the decomposition process is sufficient to collect all primitive events. Figure 4.8 presents the decomposition algorithm for EX. The algorithm, *DecompEX()* simply goes through a list of events which comprises the EX to find out if there are any composite events (CE) components. If CE exists in EX, then *DecompCE()* (shown in Figure 4.9) is invoked to recursively construct the list of primitive events that compose the original CE event. Therefore, *DecompEX()* returns *SubFilList* which contains all primitive events in the EX. The *SubfiltersConstructorAndDistributor()* algorithm in Figure 4.10 utilizes *DecompEX()* and *PrimEventToLMA()* to construct a list of all primitive events composing an EX and the corresponding LMAs for each event. This list, *EXAllocList*, is then disseminated to the concerned LMAs along with the filter name. The filter name is required to keep different instances of the same events used in more than one filter separately. Notice that every filter can have its own instance of an event because of the possibility of getting different attribute overloading for the same event. Also, the filter name is needed by DMAs to determine which DMA(s) these primitive events (in *EXAllocList*) must be forwarded to.

Input: An event expression (EX)
Output: A list of primitive events, *SubFilters*
External Functions: *IsPrimEvent*(event): returns *true* if *event* is a primitive
DecompCE(CE): returns a list of primitive events that form the composite event *CE*

DecompEX(EX)
SubFilList = Φ
for i **from** 1 **to** $|EX|$ **do**
if (*IsPrimEvent*(EX[i])) **then**
SubFilList = EX[i] $\bigcup$ SubFilList
else
SubFilList = *DecompCE*(EX[i]) $\bigcup$ SubFilList
end if
end for
return SubFilList

Fig. 4.8. Event Decomposition Algorithm.

Event Expression Processing Algorithms

The algorithms described above, enable decomposing and distribution of the monitoring tasks among LMAs. However, in order to detect the event correlation, EX and FX must be evaluated. Hence, a DMA must be selected to receive LMAs notifications and evaluate the EX of the decomposed filter. Figure 4.11 presents the algorithm that selects a DMA and allocates the EX. The selection criteria of a DMA must comply with the hierarchical management protocols explained in Section 4.2. This means DMA is selected such that LMAs would only communicate with their *containing set.* Therefore, the DMA candidates must be within the common set of all LMAs participating in detecting this filter (EX) (*CommonDMA* in the algorithm). One naive approach is to select the *highest* DMA in *CommonDMA* for evaluating the EX. This implies a centralized monitoring approach since

Input: An composite event (CE)
Output: A list of primitive events composing CE, *CEPrimEvents*
External Functions: *IsPrimEvent*(event): returns *true* if *event* is a primitive event
GetPrimEvents(CE): returns a list of events
(primitive or composite) forming *CE*

DecompCE(CE)
CEPrimEvents = Φ
List = *GetPrimEvents*(CE)
for *i* **from** 1 **to** |*List*| **do**
if (*IsPrimEv*(List[i])) **then**
CEPrimEvents = CEPrimEvents $\bigcup$ List[i]
else
CEPrimEvents = *DecompCE*(List[i]) $\bigcup$ CEPrimEvents
end if
end for
return CEPrimEvents

Fig. 4.9. Composite Events Decomposition Algorithm.

the highest common containing DMA is the DMA_{root}. Another approach is to select a *lowest* DMA in the containing set. However, this is also not a correct approach because a lowest common DMA may not be unique and thereby does not contain all LMAs. For example, if event E_1 and E_2 are both from ODU and VB (see example in Figure 3.4), then it is not sufficient for LMAs in these domains to communicate with DMAs in either ODU or VB, but rather they communicate with VA_{State} because it covers all LMAs. Also if the lowest DMA is not unique in its level, it violates the management protocol because other DMAs in the same level have to communicate with each other. Thus, the proper DMA is the *lowest* and *singleton* (based on its level) DMA in *CommonDMA*. The singleton DMAs are unique in their level and represented in *SingletonSet* in the algorithm. At the end of this process, the algorithm utilizes *Distribute()* to disseminate the information of the elected DMA to all concerned LMAs (*EXLMAs* in the algorithm). The filter name is

Input:	An event expression, *EX*, its *FilterName*
Output:	All LMAs involved in evaluating this EX, *EXLMAs*
Variables:	*EXAllocList*: Array of lists contains a list of PE, *PEname*, and their LMAs (locations), *EXLMAs*, as a list of LMAs
External Functions:	*Distribute*(Dest,<msg>): Sends the *msg* to *Dest*

SubfiltersConstructorAndDistributor(EX, FilterName)

```
EXAllocList = Φ
EXSubFilList= DecompEX(EX)
for i from 1 to |EXSubFilList| do
  EXAllocList[i].PEname = EXSubFilList[i]
  EXAllocList[i].LMA = Φ
  SubFilLMAs = PrimEventToLMA(EXSubFilList[i]);
  EXAllocList[i].LMA= SubFilLMAs ⋃ EXAllocList[i].LMA
  EXLMAs = SubFilLMAs ⋃ EXLMAs
end for
Distribute(EXLMAs,<EXAllocList, FilterName>)
return EXLMAs
```

Fig. 4.10. Subfilters Constructor and Distributor Algorithm.

also sent in the same multicast message so LMAs can figure out which primitive events are to be forwarded to this particular DMA. Remember that this correspondence can easily be established since primitive events in LMAs are associated with filter names (see Figure 4.10).

Input: *EXLMAs* produced by Algorithm 4.10 and the filter name

Output: The location of DMA that evaluates EX of *FilterName*

External Functions: *GetContDom*(LMA): returns the *containing set* for an *LMA*

Level(MA): returns the hight of *MA* in the hierarchy tree

EventExprAlloc(EXLMAs, FilterName)

```
j = 2; SingletonList = Φ; CommonDMA = GetContDom(EXLMAs[1])
for i from 2 to |EXLMAs| − 1 do
  CommonDMA = GetContDom(EXLMAs[i]) ⋃ CommonDMA
end for
for i from 1 to |CommonDMA| do
  while (j ≤ |CommonDMA| ∧
    Level(CommonDMA[i]) ≠ Level(CommonDem[j])) do
    if (j+1 = i) then
      j = j+2
    else
      j = j+1
    end if
  end while
  if (j ≥ |CommonDMA|) then
    SingletonList = CommonDMA[i] ⋃ SingletonList
  end if
end for
Minimum = Level(CommonDMA[1]); DMA = CommonDMA[1]
for i from 2 to |SingletonList| do
  if (Level(CommonDMA[i]) < Minimum) then
    Minimum = Level(CommonDMA[i]); DMA = CommonDMA[i]
  end if
end for
Distribute(EXLMAs,<DMA, FilterName>)
return DMA
```

Fig. 4.11. Event Expression Allocation Algorithm.

Filter Expression Processing Algorithms

The final step in decomposing and allocation of a filter program is the processing of the filter expression. A filter expression consists of predicates (P_i) linked in a logical expression. In order to describe the algorithm concisely, the following formalization is introduced:

Definition: A filter expression is of a set of *predicates* joined by *binary relations* (AND, and OR). In other words, $FX = (P, L)$ where P is a set of predicates (P_i) and $L = \{AND, OR\}$.

Definition: A predicate (P_i) in P is set of *right attribute* (*RAtt*), *left attribute* (*LAtt*), and a *relation* (R) such that $\forall P_i \in P, P_i = (\{LAtt, RAtt\}, \{R\}) and R = \{<, >, =, \leq, \geq\}$. We denote the right attribute and left attribute of the P_i by $RAtt_{P_i}$ and $LAtt_{P_i}$, respectively. We also use Att_{P_i} to denote either $RAtt_{P_i}$ or $LAtt_{P_i}$.

Definition: An attribute (Att_{P_i}) is a set of two elements: an *event name*, denoted by *Event*, and an *attribute name*, denoted by *AttName*. In other words, $\forall Att_{P_i} \in P_i, Att_{P_i} = \{Event, AttName\}$. Therefore, $Event_{Att_{P_i}}$ and $AttName_{Att_{P_i}}$ are used to mean the event name and the attribute name of the right or left attribute of predicate P_i. Similarly, $Event_{RAtt_{P_i}}$, for example, can also be used to denote the event name of the *right attribute* of the predicate P_i.

The algorithm presented in Figure 4.12 is used for decomposing the monitoring demand expressed in a filter expression (FX) into content-based and correlation-based filtering tasks (subfilters) performed by LMAs and DMAs, respectively. The algorithm also allocates these filtering tasks to MAs based on the environment specifications (i.e., events locations) and then disseminates the monitoring tasks to the required MAs. First, the algorithm uses previous algorithms to convert every composite event in FX to a set of its primitive events (line 2 and 3). Then, each decomposed CE associated with an attribute in FX is replaced with the corresponding primitive event that contains this attribute name, $AttName_{Att_{P_i}}$ (line 5 to 7). This goes for all predicates (P_i) in FX (line 2 through 13) for both left and right attributes of a predicate. So by the end of line 13, $FX_{decomposed}$ is

flattened into primitive events only which are stored in *DecompFX*.

The next step in the algorithm is to allocate and distribute these predicates among MA based on the complexity of these attributes. For example, if Att_{P_i} has a constant (VALUE) such as a number or a string in a left attribute, P_i represents a primitive event which can be handled by LMAs. Similarly, if the right attribute and the left attribute belong to the same event, then this predicate, P_i, also represents a primitive event that can be allocated to one or more LMAs (line 15). In both cases, the event of P_i ($Event_{RAtt_{P_i}}$) is used to determine that the proper LMA set using *PrimEventToLMA()* and then the P_i and the filter name are disseminated to the corresponding LMA set as shown in line 16 and 17. The filter name is included in the multicast message for the same reason described in the previous algorithm (see Figure 4.8). All LMAs that are assigned P_i (subfilter) as a delegation are collected in FXLMAs (line 18) which is used later (line 26) to send them the name DMA selected for FX evaluation. If the predicate P_i contains two different events, then this P_i is a correlation and can only be delegated to a DMA (line 19), In line 20 and 21, the algorithm assembles a set of locations for all primitive events composing correlation predicates (CompositeFXPred). This set is used later in line 24 to elect the DMA that evaluates the $FX_{decomposed}$. Then, the $FX_{decomposed}$ and the filter name is then sent to the elected DMA (line 25) and the LMAs participating in evaluating $FX_{decomposed}$ are also informed of the DMA location (line 26). As a result of this algorithm, FX is decomposed into subfilters and a simpler correlation expression which are then allocated to LMAs and DMAs to distribute monitoring load efficiently and minimize the events propagation and intrusiveness.

Input: A filter expression, FX, and a filter name

Output: Decomposed FX and its LMAs and DMAs evaluators

FilterExprDecompAndAlloc(FX, FilterName)

```
01    DecompFX ← Φ; FX_decomposed ← FX
02    for all P_i ∈ FX_decomposed
03       if (Event_AttP_i is CE) then
04          PEList ← DecompCE(Event_AttP_i)
05          for all PE_j ∈ PEList
06             if (AttName_AttP_i ∈ PE_j) then
07                Event_AttP_i ← Event_PEj; DecompFX= Event_PEj ⋃ DecompFX
08             end if
09          end for
10       else
11          DecompFX = Event_AttP_i ⋃ DecompFX
12       end if
13    end for
14    for all P_i ∈ DecompFX
15       if (RAtt_Pi is VALUE ∨ Event_RAttP_i = Event_LAttP_i) then
16          LMAs = PrimEventToLMA(Event_RAttP_i)
17          Distribute(LMAs,<FilterName,P_i >); FXLMAs= LMAs ⋃ FXLMAs
18          FXLMAs = LMAs ⋃ FXLMAs
19       else
20          CompositeFXPred=PrimEventToLMA(Event_LAttP_i) ⋃
21             PrimEventToLMA(Event_RAttP_i) ⋃ CompositeFXPred
22       end if
23    end for
24    FXDMA ← EventExprAlloc(CompositeFXPred)
25    Distribute(FXDMA, <FilterName, FX_decomposed >)
26    Distribute(FXLMAs, <FilterName, FXDMA>)
27    return FX_decomposed, FXLMAs, FXDMA
```

Fig. 4.12. Filter Expression Decomposition and Allocation Algorithm.

Filter Decomposition Optimization Techniques

In the previous algorithms, EX and FX are allocated in one DMA. However, in some cases, allocating EX and/or FX among a number of DMAs could be more efficient. For example, if the EX/FX is too long and contains events from different locations in the application environment, decomposing/fragmenting EX/FX itself into subexpressions, where each subexpression is assigned to the proper DMA, could produce a more efficient monitoring and filtering mechanism. This technique distributes the monitoring load further and circumscribes the events propagation more within the domains bounds. On the other hand, due to the DMAs communication overhead generated from exchanging the subexpression results, the monitoring latency may increase. For this reason and since the network latency is a dynamic parameter which is best managed by users themselves, we delegate the responsibility of enabling or disabling the optimization technique to the users/consumers for each filter. In the following, we outline the main steps of optimizing the decomposition and allocating process for FX an EX (we use Expr to denote both FX and EX).

Every boolean expression over {+, . , -} boolean operators[§] can be written in disjunctive or conjunctive normal form[¶]. The algorithm basically divide the expression into set conjunctive (i.e., AND) subexpressions joined in disjunctive operations (i.e., OR). Each of these conjunctive subexpressions is assigned to the proper DMA using the algorithm in Figure 4.12. Then, the global DMA that evaluates the disjunctive expression is elected and announces to the DMAs participating in evaluating the conjunctive components. One advantage of this allocation is that one true conjunctive subexpression is enough to make the EX/FX true. This further implies that forwarding notifications and exchanging monitoring information among the DMAs is minimized. This is in addition to utilizing the concurrent DMA evaluation of EX/FX which increases the system performance and reduces the monitoring latency.

Subscription Protocol

As discussed in Chapter 3, consumers may add, modify or delete filters on-the-fly through the manager interface. When adding a new filter, it must have a unique filter name.

[§]They, respectively, correspond to OR, AND and complement boolean algebra operators.

[¶]Theorem in [64] page 591.

Input: A filter or event expression and a filter name

Output: A decomposed expression in disjunctive normal form

OptimizedExprDecompandAlloc(Expr, FilterName)

1. (Initialization.) ExprDMAs $\leftarrow \Phi$
2. (Convert to EX/FX Disjunctive Normal Form.) The *Expr* is converted into a disjunctive of conjunctive expressions [31], ($Conj_i$), and the resulted expression is called $Expr_{fragmented}$
3. (Decompose and allocate each conjunctive.)
   for all $Conj_i \in Expr_{fragmented}$

   $DMA_{Conj_i} \leftarrow FilterExprDecompAndAlloc(Conj_i$, FilterName)

   ExprDMAs = $DMA_{Conj_i} \bigcup$ ExprDMAs

   end for
4. (Allocate and Distribute the disjunctive.)
   (a) Using *ExprDMAs* set, we can find out the containing sets for each DMA and then select the *proper* DMA, called *global DMA*, as described in the algorithm in Figure 4.8.
   (b) The *global DMA* is assigned to evaluate the disjunctive expression.
   (c) The location of the *global DMA* is disseminated to all DMAs participating in evaluating the conjunctive expressions, so that the results are forwarded to the *global DMA*.
5. (Return expression in DNF) **Return** $Expr_{decomposed}$

Fig. 4.13. Expression Decomposition and Allocation Optimization Algorithm.

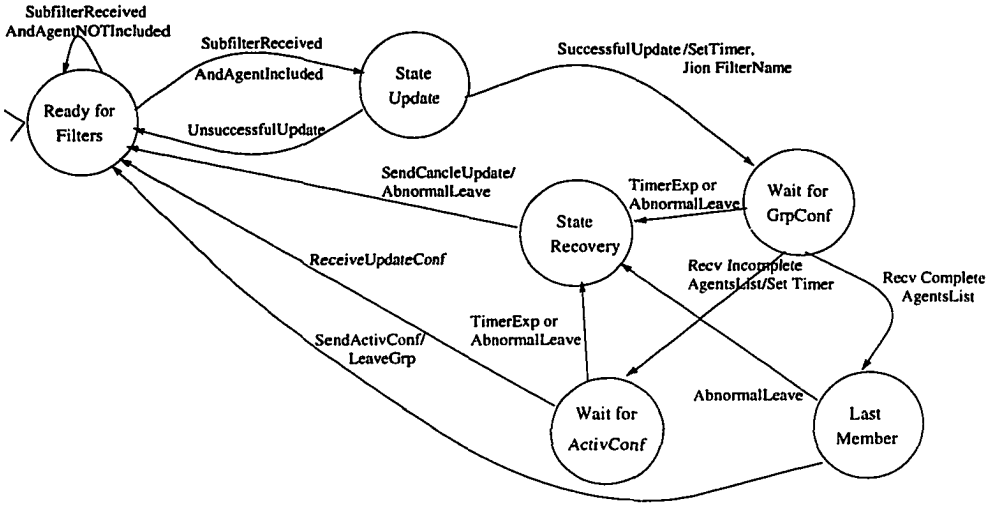


Fig. 4.14. Subscription Protocol State Diagram.

However, when modifying or deleting a filter program, it must match a filter name used in the monitoring system (i.e., an existing filter). Adding, modifying or deleting filters may create an inconsistency in the monitoring environment. This can be resolved by using the *subscription protocol* depicted in Figure 4.14.

The subscription component parses, decomposes the filter program, and sends to the monitoring agents a multicast message (*Subfilter*) that includes a *filter name*, *subfilters*, and a *list of agents* involved in this monitoring task. If the Subfilter messages received contains the agent ID ($MachineName.DomainName$), the agents then perform the filter composition to insert the subfilter information into the DAG or PN (*Update State*). If an agent state is updated successfully, it joins (using RMS) a multicast group, `FilterName`, included in the message itself and waits for the join confirmation from RMS. The join conformation contains a list of all members (agents) in the group. In RMS, the join operation and conformation are performed as one atomic action based on the token ring protocol [4]. In this case, the members list in the join confirmation indicates agents in the group which have successfully modified their states so far. Thus, when an agent discovers that all concerned agents have joined the group, it sends a multicast message (*ActiviateConf*) to the group to activate the received filter and it then leaves the `FilterName` group. On the other

hand, if an agent fails to update its state or to join the group for `TimeOut` period, then the first agent that times out (timer expires), sends a multicast message to the `FilterName` group to cancel and recover the state update. Every agent sets up a timer right after receiving the join confirmation message for $TimeOut \geq (N+1) * RTT + MRT$ such that RTT is the maximum round trip delay in the network, MRT is the maximum packet retransmission time in RMS, and N is the number of agents contained in this subfilter message. RTT is used to calculate the join notification time, and MRT is used to accommodate the network delays and retransmissions. Neglecting the state update time, both factors must be considered, otherwise, agents may time out before the protocol operation completes. Agents receive an automatic notification from RMS when an agent withdraws from the group because of normal (e.g, leave) or abnormal events (e.g., agent crashes). If this occurs before receiving *ActiveConf* message, then agents cancel their state updates and quit the process. At the end of this operation, consumers get notified about the results of their subscription, (e.g., confirmed or aborted) by a monitoring agent. This protocols is important for assuring agents consistency as well as synchronizing the agents monitoring operations. Simplicity and minimal overhead are major advantages of this protocol compared to other distributed algorithms such as two-phase commit protocol [88].

4.3.5 Event Detection

The MAs receive the delegated monitoring tasks (subfilters) [30] and configure themselves accordingly by inserting these subfilters in its filtering internal representation, such as the direct acyclic graph (DAG) [12] or Petri Nets (PN) [27]. This process is called filter composition [3] and is described in detail in Section 5.3. The LMAs and DMAs in the *monitoring agent network* (see Figure 4.4) then work cooperatively based on the management protocols described in Section 4.2 for monitoring the target application using distributed subscription requests (filters). Section 5.3 provides more discussion on event detection techniques used by LMA and DMA.

4.3.6 Monitoring Action

This represent the final step in the monitoring process. If an interesting event pattern is detected, the monitoring system performs the corresponding action defined in High-level

Action Specification Language (HASL) described in Section 3.3. Figure 4.4 depicts the general monitoring process where detected events are sent to a *collector* that collects and analyzes the monitoring information. And the outcome results from the collector can be viewed (visualized) in the monitoring presentation device.

Analyzing, presenting and visualizing the monitoring information are not in the scope of work of this thesis, however, our monitoring architecture enables the users to integrate the analysis and re-action services in the monitoring process itself by using the HASL and the Control Component. The monitoring agents that trigger a filter perform the action associated with this filter, such as forwarding the monitoring information to the corresponding managers (see Section 3.3).

4.4 Summary

This chapter describes the algorithms and protocols used for implementing the hierarchical filtering-based monitoring architecture. This architecture identifies two classes of monitoring agents: Local Monitoring Agents (LMA) that detect primitive and Domain Monitoring Agents (DMA) that detect composite and correlated events. Monitoring agents are organized in a hierarchical structure based on the environment specification such that one or more LMAs are connected to a single DMA. DMAs are also connected to higher DMAs in the hierarchy which enables monitoring agents to detect correlation events in domain basis. In order to enhance the management and communications between monitoring agents, an optimization technique is used such that an LMA can forward notifications only to those DMAs that need such events, instead of forwarding them to the next DMA in the agent hierarchy. The monitoring agents use the reliable multicasting service (RMS) to disseminate events to a group of interested agents. In order to distributed the monitoring load in the agents hierarchy, the monitoring system (HiFi) employs *decomposition and allocation algorithms* that decompose (1) the composite events in users filters into sets of primitive events, and (2) filters expressions into subexpressions that can be evaluated in the same monitoring agents. The allocation algorithm is then used to distribute the decomposed filter information among agents based on the agents roles and the environment specifications. As a result, each agent is assigned filtering responsibilities based on its monitoring scope. The monitoring information forwarded from the monitoring agents in the hierarchy

are then integrated by a single DMA to detect event correlations. In particular, the lowest singleton common DMA of all LMAs participating in processing this filter is selected. Therefore, there is no a specific DMA assigned for detecting all filters, however, a DMA is selected based on the specification of each filter (i.e., filter events and expressions).

The *agent organization protocol* is used to setup the agents hierarchy automatically and without users intervention. This protocol performs a sequence of agents creation and election steps for building the hierarchy starting from the leafs of the hierarchy (LMAs) until the DMA_{root} is created. The monitoring agents hierarchy is dynamic and can be horizontally expanded at any level in the hierarchy to accommodate an excessive monitoring load in a DMA. The *dynamic subscription protocol* is used to resolve the agents state-inconsistency problem that results from receiving the filter information (decomposition) by agents at different time. This protocol not only guarantees an atomic state-update in the agents group, but also synchronizes the monitoring operations among the agents. This chapter also describes the advantages of employing the hierarchical monitoring architecture in improving the scalability and performance, and minimizing the intrusiveness of the monitoring system.

CHAPTER V

SYSTEM COMPONENTS AND IMPLEMENTATION

This chapter describes the design and the implementation of the monitoring systems components: *Instrumentation, Subscription Service, Event Filtering* and *Control.* While the previous Chapters focus on architectural issues including algorithms and protocols used in HiFi monitoring system, this Chapter describes the system components that performs these tasks. The implementation of each component and their interaction with event producers and consumers are also presented. Figure 5.1 shows the components of the monitoring system and their interaction. It also shows the monitoring interaction with event consumers via filters and event producers (or the monitored objects) via events. Another goal of this chapter is to explain how this implementation contributes in achieving the work objectives described in Chapter 1 such as improving performance and scalability and minimizing the intrusiveness.

5.1 Instrumentation Component

The process of inserting monitoring instructions into the code of observed programs is called the *instrumentation process.* These monitoring instructions act as sensors inside the programs body and report the events as they occurred to the LMA during the program execution. In order to monitor the execution of any application in real-time, the application must express its state by conveying all information of the occurred events. In other words, in order to provide management services for distributed systems, these systems must report events that represent their run-time behavior. The developers must also intervene to insert the monitoring instructions in the proper places (e.g. in routines under investigation). In the following, we discuss the services and functions provided by the instrumentation component and subcomponents.

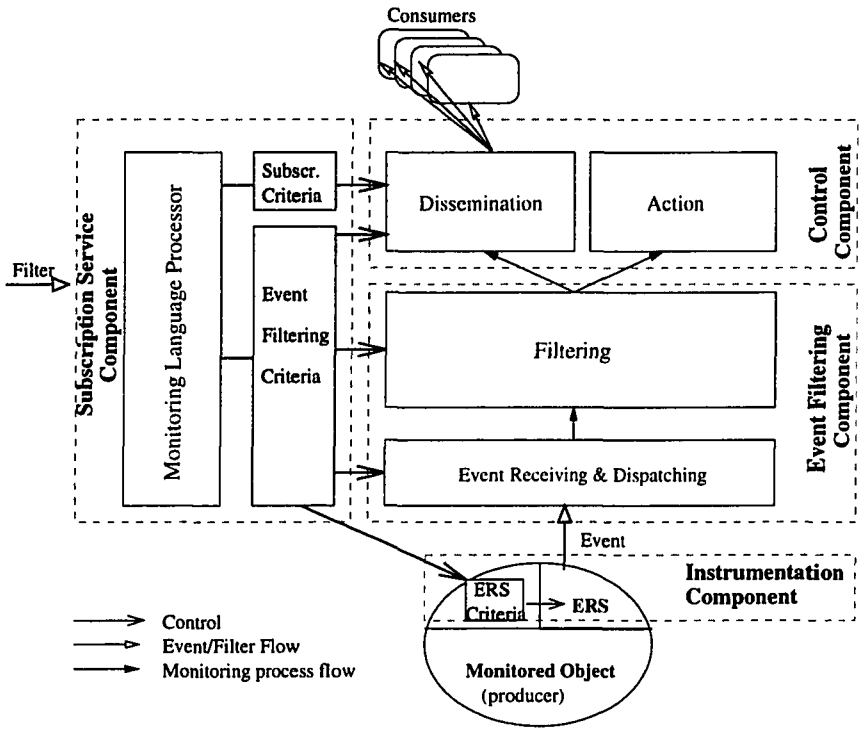


Fig. 5.1. Monitoring System Components.

5.1.1 Event Specifications

To facilitate the instrumentation process for application developers, the monitoring system provides the *Event Specifications Generator (ESG)* subcomponent to support *automatic generation* of low-level event formats from the high-level user specifications.

The event specifications (or the notification format) must be specified at an early stage and prior to any monitoring operation. The event specifications must contain all information required to recognize any particular event. The ESG enables the developers/users to define the event specifications in a high level language called High-level Event Specification Language (HESL) which is described in detail in Section 3.3. The ESG subcomponent utilizes the event information supplied by the subscription component to convert the event specifications defined in HESL into low-level event formats called Event Reporting Criteria (ERC) which is directly used in the monitoring process (see Figure 5.2).

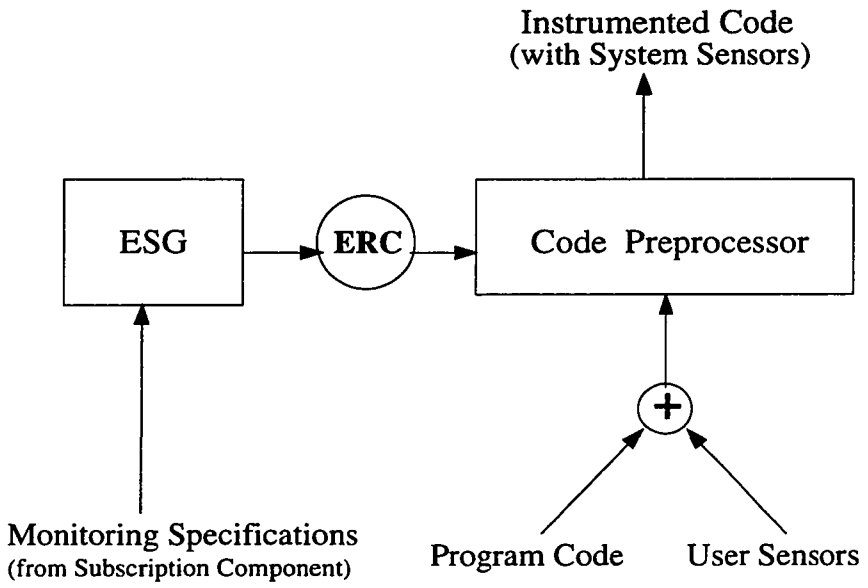


Fig. 5.2. Code Instrumentation Process.

The low-level event format (ERC) contains the event location, event domain, reporting mode, number of event attributes, the type of each attribute such as an integer, float or string. Therefore, users specify the list attribute names (and values if necessary) in a declarative way without requiring them to indicate the type of each one. The low-level format (ERC) is then used by other components for further instrumentation processing.

5.1.2 Automatic Event Insertion

The main function of the instrumentation component is to facilitate the process of inserting the monitoring instruction or sensors into the program code. In many monitoring systems [58, 69], programmers write a considerable amount of code for each generated event. This makes the instrumentation task tedious and error-prone. Furthermore, it may obscure the program's appearance and cohesion. For these reasons, one of our design principles is to make the code which needs to be inserted manually by programmers as minimal as possible. This relieves the burden of writing and maintaining the sensors code and reduce the possibility of human errors. For example, in order to generate event *RMP-*

SWarnings described in Section 3.3, users insert the following statement at the proper place of the code:

ReportEvent(*RMPSWarnings*)

We call this statement the *user sensor*. However, this does not provide sufficient information to generate an event because it does not provide information about the event attributes into the instrumented program. Thus, after users complete inserting their sensors in the code, the instrumentation component pre-processes the instrumented code and replaces user sensors with extended sensors that convey all events related information. This type of sensors is called *system sensors*. For example, the user sensor ReportEvent(*RMPSWarnings*) in the previous example is replaced by the following system sensor:

```
ReportEvent("RMPSWarnings",ModName,FuncName,"IMMEDIATE", 2,
            "EventType", STRING,"Warning","Machine", STRING,"dragon")
```

The *ModName* and *FuncName* are variables that are assigned at the beginning of the program and function, respectively. To provide more flexibility in using HESL, the assignments of these variables are left to the users. This enables mapping an event to several module names without having to define an event for each module. For example, users can use *RMPSWarning* to report *warning* events from modules other than RMPS by setting the values of ModName and FuncName variables. The third argument specifies the event reporting mode (Immediate or Delayed), and the fourth one specifies the number of attributes in the event. The rest of the arguments in the system sensor is the variable attributes names, types and values. These automatic insertion of system sensors is performed in a copy of the original source to retain the program code cohesion. The new instrumented program copy called ".HiFi_<ProgramName>". The automatic insertion process utilizes the ERC information generated by ESG to construct the system sensors.

In addition to the automatic insertion of system sensors, the instrumentation component generates a make file called *.HiFi_Makefile* based on the original make files. Therefore, users have only to remember using "-f" option with make command whenever they want to compile their programs with monitoring capabilities.

5.1.3 Dynamic Event Signaling

Event signaling is the process of *constructing* and *sending* the event notification message. This task is performed by the *Event Reporting Stub (ERS)* which is a supported library or stub linked with the monitoring application in order to construct and report the occurrence of events.

When the instrumented program generates an event, it invokes `ReportEvent()` which is a unbounded parameters-list function that ERS uses to construct the notification message and send it to the connecting LMA. The code is shown in Figure 5.3. Events are dynamically activated or deactivated as a consequence of adding or deleting filters at run time. The filter activation and deactivation information are sent to LMAs during the filter decomposition and allocation. Then each LMA notifies the associated ERS which updates the *Event Vector Table (EVT)* accordingly. The *EVT* is basically a table that lists the defined events and their corresponding flags which indicate if an event is active or inactive. In other words, EVT is configured by the subscription component to activate or deactivate reporting events. The developers or users may define and insert all kinds of events in the program. However, ERS in the instrumentation component determines the *active* events according to the subscription demands and reports only events of interest at any particular time. This highly reduces the number of reported and processed events and allows the developers to dynamically select events to be reported at run-time. Therefore, this significantly reduces the monitoring intrusiveness without sacrificing the flexibility and ease of use of the monitoring system.

One disadvantage of the dynamic signaling is that it requires LMA to communicate back with ERS which could increase the monitoring intrusiveness and program perturbation. In some critical applications, event set are mostly static and the overhead of delegating this filter task to LMA is less than the overhead incurs by the dynamic signaling. For this reason, the user can choose to enable and disable the dynamic feature by invoking this function in ERS: `ERS->DynamicSignalling(ONOFF)` at the beginning of program execution. Furthermore, in order to minimize the overhead resulting from dynamic signaling and to avoid integrating the ERS events in the program event loop, ERS is designed such that it checks the event status (such as activating new events) only by probing its LMA and after `ReportEvent()` is invoked by the program itself. More specif-

```
int ReportEvent(char *Ev,char *Mod,char* Fun,char *Rep, int cnt, ...)
{  /* Fix Attributes */
  strcpy(event->evname,Ev);
  if (ON)
  {  /* Check Event in Vector Table */
    if (CheckEventStatus(event->evname) == INACTIVE)
    { delete event; return 0;     }
  }
  strcpy(event->mod,Mod);
  strcpy(event->func,Fun);
  strcpy(event->rep, Rep);
  strcpy(event->id,Ev); /* could be evname */
  /* Variable Attributes */
  va_start(ap,cnt);
  event->PredCount=cnt;
  for (i=0; i < cnt; i++)
  {
    strcpy(event->Predicates[i].name, va_arg(ap,char*));
    event->Predicates[i].rel=1;
    event->Predicates[i].type=va_arg(ap,int);
    switch (event->Predicates[i].type)   {
    case INTEGER:
      event->Predicates[i].value.intv=va_arg(ap,int); break;
    case FLOAT:
      event->Predicates[i].value.fltv=va_arg(ap,double); break;
    case STRING:    {
      int j=0; char svalue[32];
      for(j=0,sval=va_arg(ap,char*); *sval;sval++,j++) svalue[j]=*sval;
      svalue[j]='\0';
      strcpy(event->Predicates[i].value.strv, svalue); break;
    }
    default:
      cout<<"<<ERS ERROR>> Invalid Parameter Type=%d! .. \n";
      return -1;
      break;
    } // end of switch
  } // end of for
  va_end(ap);
```

```
  if (!strcmp(event->rep,"Immediate"))
  {
    EnQueue(event);
    SendEventQueue();
  }
  else if (!strcmp(event->rep,"Delayed"))
  {
    Limit=EnQueue(event);
    if (Limit >= MAX_EQ_LENGHT)
    {
      SendEventQueue();
      EmptyEventQueue();
    }
  }
  else
  { cout <<"<<ERS Error>> Invalid Report Mode"<< endl;   return(-1);  }

  if(usock->SendData(LMAsock, (BYTE *) event, sizeof(PrimEventNotif)) < 0)
  {  perror("send"); exit(0); }
  delete event;

  if ((LMAMsg=CheckMailBox(LMAsock)) != NULL)
  {
    UpdateEventVectorTable(LMAMsg);
    return 1;
  }
  else /* No msgs from LMA */
    return 0;
}
```

Fig. 5.3. ReportEvent in ERS.

ically, sending an event from ERS represents a signal for LMA to down-load any event activation updates, if any. ERS checks the communication buffers for some timeout period right after the event is sent using `CheckMailBox()` function (see the code in Figure 5.3). If an event update is received, then ERS invokes `UpdateEventVectorTable()` to modify the EVT accordingly. This enables the consumers to select or suppress events at run-time without interfering with program execution (i.e., UNIX `select()` system call is not used for this purpose)

5.1.4 Adjustable Event Reporting

The rate of event delivery can be controlled via the adjustable reporting mechanism supported in ERS. Consumers can delay reporting the event notifications till a specific time, which is called *delayed reporting*, or request an immediate notification of the event occurrence, which is called *immediate reporting*. Therefore, the `Mode` attribute in the event specification can be either `Immediate` in which the event will be generated and sent to the monitoring system right after its occurrence, or `Delayed` in which events are buffered until one of the following condition occurs: (1) an immediate event occurred, (2) the program invokes `FlushEvents()` which is a service provided in ERS, (3) the number of buffered events exceeds the maximum threshold allowed by the LMAs. We deliberately avoid using time limits as one of the options to avoid any perturbation caused by timers interrupts. However, consumers/programmers can still set up their own timers in the program and flush the event buffer using the `FlushEvents()` service call.

The delayed event reporting mode is used to accumulate more events and minimize the *send* frequency and thereby reducing the intrusiveness of the monitoring system. On the other hand, the immediate event reporting mode is used when events are time-critical and real-time event notification is required. The adjustable reporting feature enables users to control the monitoring intrusiveness and the *event freshness* trade-off.

5.1.5 Automatic Monitoring Agent Creation

One of the services provided by ERS is to participate in setting up the agent hierarchy automatically. ERS provide the `ERSInit()` function which starts RMPS and LMA, and then connects to LMA using UNIX sockets. ERS issue this connection after LMA send

a ready signal. After the connection is accepted by LMA and the final agents setup confirmation is received, ERS waits for the event information from the LMA to update the event vector table. The detailed description of the agent's hierarchical setup and allocation is described in Section 4.3.3. Thus, `ERSInit()` must be the first statement in an instrumented program. In some case, several application processes share one LMA in the same machine. In this case, it the responsibility of users/programmers to insert ERSInit() in the first process to start and `ConnectToLMA()` in all other processes. In other words, only one process (the leading one) is required to use `ERSInit()` and other processes use `ConnectToLMA()`. This architecture enables the users to customize the agent structure based on the application requirements and control the trade-off of LMA per process or LMA per machine. Only after ERS receives the LMA confirmation and the event information, it resumes the application execution. Otherwise, it terminates the application program.

5.2 Subscription Service Component

The monitoring subscription is the process by which the consumers would specify their monitoring demands represented in filter programs using HFSL. The filter is basically represented by an event expression, filter expression and action specification. Therefore, defining the event, action and environment specification is in fact part of the subscription process. Section 3.3 presents a detailed discussion on the formal specification of the monitoring system language (MSL) composed of HFSL, HESL, HASL and ESL.

The subscription service component is used to provide the following main services:

- Processing users specifications and demands such as adding, deleting or modifying monitoring requests
- Receiving and processing monitoring results such as forwarded events and notification
- Controlling monitoring agents' activities such as start, stop or reset agents

In order to support such services in our monitoring system, the subscription component performs the following tasks:

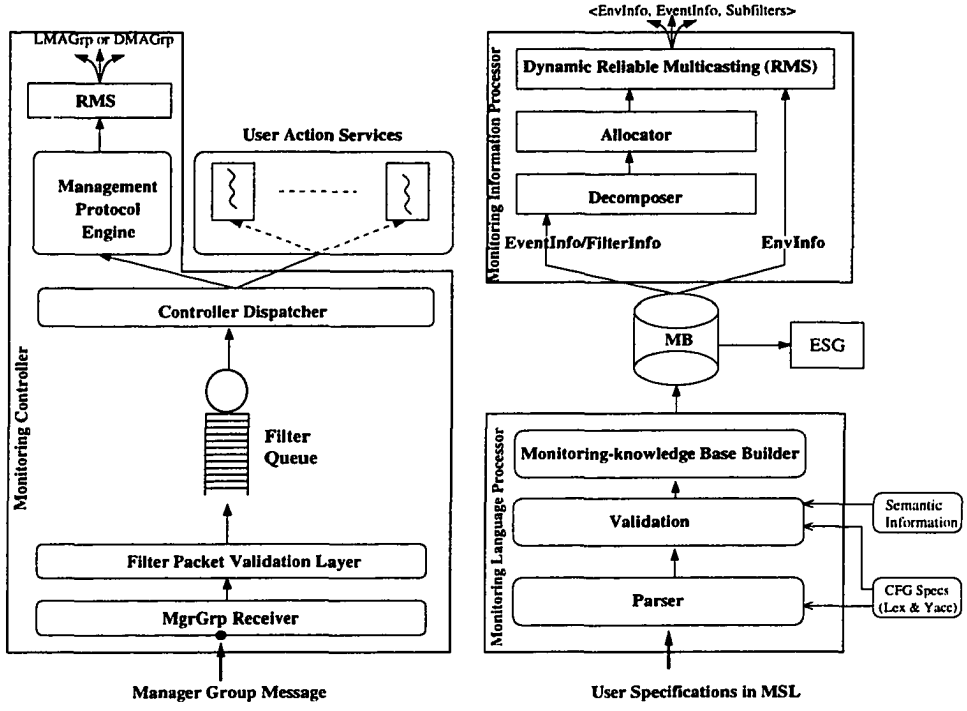


Fig. 5.4. Subscription Component.

- Parsing and validating the user specifications
- Constructing the Monitoring-knowledge Base (MB) information
- Decomposing events and filters specification
- Distributing monitoring tasks
- Processing monitoring results information received from MAs
- Performing the agents management protocols

As shown in Figure 5.4, the subscription service component has three major subcomponents: *Monitoring Language Processor (MLP)*, *Monitoring Information Processor (MLP)*, and *Monitoring Controller (MC)*. The following sections describe each of these subcomponents and the corresponding tasks.

5.2.1 Monitoring Language Processor

The Monitoring Language Processor or MLP is used to support the first service of the subscription component which is reading and processing users specifications. Users specifications comprises event, filter, action and environment information specified in HESL, HFSL, HASL and ESL, respectively. It also includes adding, deleting or modifying filter programs. MLP reads users specifications from the standard input or from a file, and then performs the following assigned tasks:

Parsing and Syntax Validation: Scanning and parsing users specifications is the first task performed by the scanner and parser in the MLP, respectively. The scanners begins the analysis of the user specification by reading the input, character by character, and grouping characters into individual words and symbols called *tokens*. The parser reads these tokens and groups them into units as specified in the *production rules* of a context-free grammar (CFG) [23, 37] supplied to the parser. *Lex* and *Yacc* [51] are used for implementing the scanner and parser functions in MLP. Lex is a lexical analyzer generator that recognizes a regular expression, but Yacc is LALR(1) parser that generates a C-program implementing the state machine of the CFG provided by the user. The CFG of the monitoring language is presented in Section 3.3. LALR parsers combines the advantages of LR and SLR parsers which produces a better resolution and smaller tables. The monitoring language (HESL, HFSL, HASL, and ESL) production rules are written and aligned in Yacc format such that no ambiguity or conflicts (e.g., shift/reduce or reduce/reduce) exist. Throughout our discussion, we use "parsing" to mean both scanning and parsing at the same time. The parser in MLP is essential to validate the syntax of the user specification and to provide the means to extract the information required to build the monitoring-knowledge base and validate the semantic of the user specifications. If a syntax error is encountered in the user specifications, the program terminates at the error location in the user specifications.

Semantic Validation: The monitoring system performs some semantic checking during the parsing process described above. There is no formal semantic specification used for this purpose since this is not the focus of this thesis, however, some semantic rules are

provided to MLP to check out during and after the parsing process. Examples of such semantic rules used in the monitoring language includes the following:

- *HESL Semantics Rules*
  - Event names must be unique
  - Module names must appear in the ESL
  - Fixed attributes must be specified
- *HFSL Semantics Rules*
  - Events in the filter expression must be included in the event expression
  - Attributes in the filter expression must be contained in the event definition
  - Attribute types must match correctly
- *ESL Semantics Rules*
  - Domain must not contain itself or any of its containing domains
  - Modules names used in HESL must appear at least once in ESL
  - Every machine must be contained in a one domain only

These are examples of semantics rules that MLP uses to validate the user monitoring specifications. If invalid semantic is discovered, users are prompted with explanation of the user specifications violation.

Constructing Monitoring-knowledge Base: One of the major tasks performed by MLP of the subscription component is constructing the *Monitoring-knowledge Base (MB)*. The MB information is then used by the Monitoring Information Processor (MIP) for pursuing further tasks such as event/filter decomposition and allocation which are described in details in Section 4.3. The MB Builder in the MLP constructs the knowledge base structure and information. MB Builder receives the information from the parser after parsing and validation and converts it into tables of linked lists that are easily accessed and fetched by other monitoring utilities. MB looks like a database for retrieving all information related to a specific monitoring application/operation. For example, number of machines

involved, where every machine is located, how domains are related are all query examples that can be answered by a specific MB. Since MB does normally consume large memory space, MB is constantly available and can be dynamically fetched at run-time. Figure 5.4 shows that the ESG in the instrumentation component uses the MB information to create ERC that then used for inserting the system sensors as described in the Section 5.1.

The MB is completely based on list data structures using an extended version of widely used `List()` and `ListItr()` classes. The main advantage of using such Abstract Data Type (ADT) is to optimize memory usage and improve the program maintainability. Appendix B.2 and Appendix B.3 show the primitive and composite event data structure, and Appendix B.1 and Appendix B.4 show the environment and filter data structure which are both used to construct tables events, environment and filters tables, respectively.

Dynamic User Subscription: Dynamic user subscription that enables adding, deleting or modifying filter programs at run-time is one of the MLP functions. This typically involves reading, parsing, validating, decomposing and then distributing the HFSL of a filter as described before. The new added filter must assign a unique name in the monitoring system. On the other hand, deleting a filter requires only multicasting this request to LMAGrp and DMAGrp. The monitoring agents, consequently, evict all predicates belong to this filter in the DAG or PN and deactivate events are used with this filter only. The deleted filter name must match an existing filter in the monitoring system. The filter modification process is composed of deleting this filter and adding the new modified one. In adding, deleting or modifying filers, managers and agents comply with the dynamic subscription protocols described in Section4.3.4 in order to maintain consistency and synchronize agents' activities.

5.2.2 Monitoring Information Processor

The *Monitoring Information Processor (MIP)* subcomponent produces the monitoring information in a format usable by the monitoring agents. This involves (1) retrieving and combining different information from MB, (2) decomposing and allocating such information to generate monitoring tasks executable by the monitoring agents, and (3) packaging and disseminating the produced information to the agents using RMS. The first part of this information is the environment information (EnvInfo) which is used for establishing

the agents hierarchy as described in Section 4.3. The EnvInfo contains a list of records where each record states the domain/superdomain name, unique ID and list of domains and/or machines included in each domain and superdomain. This information is used by LMAs and DMAs in the election process such that the first element (machine or domain) represents the LMA/DMA leader of this agent group (see Section 4.3.3). After the agent hierarchy is constructed, the EnvInfo is disseminated to LMAGrp and DMAGrp groups. On the other hand, MIP constructs the EventInfo and FiltersInfo tables from MB and passes them to the *composer* and *allocator*, as shown in Figure 5.4, to generate decomposed subfilters and disseminate this information dynamically to the associated agents. Decomposition and Allocation algorithms are described in Section 4.3.

5.2.3 Monitoring Controller

The *Monitoring Controller (MC)* is the subcomponent responsible for receiving and processing incoming messages sent to the manager program via the MgrGrp multicast group. There are two types of messages: (1) monitoring information results that are forwarded from MAs to indicate event discovery, and (2) monitoring control messages which are sent by the MAs to perform certain management protocol. If it is the former case, the received monitoring information is dispatched to the corresponding service routine that matches the filter name in this message. These services are specified by the user during the subscription process and are activated according to the detected filter. On the other hand, the control messages is forwarded to the *management protocol engine* (see Figure 5.4) which parses and performs the corresponding protocol action. Examples of such protocols include automatic agents allocation and dynamic subscription protocol as described in Section 4.3.

5.3 Event Filtering Component

The event filtering component is the core component of the monitoring system. Its main functionality is (1) receiving and processing filtering tasks delegated from the subscription component after decomposition and allocation, and (2) inspecting incoming events based on the event attributes and the filter information (i.e., filter internal representation) to determine if this event is interesting (*detected*) or irrelevant (*rejected*). The former is

performed by the *subfilter processor* in a process called filter composition but the latter is performed by the *event processor* subcomponent in a process called event filtering. They both operate on the *event filtering internal representation* that represent the monitoring information such as consumers' subscriptions and event specifications. The subfilter processor, event processor and event filtering internal representation constitute the internal architecture of the monitoring agents, LMA or DMA.

5.3.1 Event Filtering Internal Representation

The internal representation of filters is a key issue in designing event filtering mechanisms. The filter internal representation determines the model and the data structure used to express the meaning of a filter. In Section 2.2, we discussed various filtering internal representations used by other event filtering mechanisms. Our architecture integrates two different representations: Directed Acyclic Graph (DAG) and Petri Nets (PN) into one framework to enable detecting both primitive and composite events. Previous work in event filtering are polarized toward one model over another which results in a half solution of the problem of monitoring distributed systems.

DAG Implementation: The DAG representation is used by LMAs to match primitive events. LMAs are stateless agents since they are responsible for detecting primitive events only which do not require storing event history or tracking composite events. However, DMAs use a filtering internal representation that has the capability to store and memorize the event history such as DFA (Deterministic Finite Automata) and PN (Petri Nets) representation because a DMA is responsible for detecting composite events. The DAG filtering representation consists of nodes connected by edges in an acyclic graph. Its nodes represent the test predicates and the edges represent the control transfer. The DAG is parsed top-down such that if the test predicate is true, the right-hand edge is traversed, otherwise the left-hand edge is traversed. Thus, the evaluation result of the test predicate (either *true* or *false*) determines the edge to traverse. An event is *detected* if the terminating node (leaf node) is denoted as *true*. There are two terminal nodes in the graph, the *true* node that denotes the acceptance of the event and the *false* node that denotes the event rejection.

In order to optimize memory usage, we implemented the filter DAG using linked

lists data structures. Appendix C.1 illustrates the basic data structures used for implementing DAG filters representation. In our system, the DAG is implemented as a list of nodes (DAGNode) indexed by module names (ModName). Each node consists of a list of node configurations (DAGNodeInfoTable) where each configuration indicates a unique function name and list of predicates (Pred) indexed by this designated function name. The predicate (Pred) is a list of attributes (ATTRIBUTE) which consists of attribute name (name) and list of values (Value). An attribute may take multiple values because different filters may specify different values for the same attribute name. The *Item* is the same as *RValue* in Appendix C.2. The DAG nodes and structure are processed using indexed hash tables to provide a faster access. Unlike DAG implementation in previous filtering work [12, 59, 63, 94], this indexed hashing implementation enables the DAG iterators to search the DAG for matching events in less time than traditional binary nodes DAG implementation used in these filtering mechanisms. We will discuss this issue in more details in Section 5.3.4.

PN Implementation: The general structure of Petri Nets is described in Section 2.2. PN representation is needed for storing the event history that can be later used in composite event detection [27]. Detecting composite events requires information on two or more events that had occurred in the system. Deterministic Finite Automata (DFA), as described in Section 2.2, can also be used for this purpose [29]. However, we decided to use PN in representing event correlation rather than DFA for the following reasons [27]:

- PN model is more powerful for representing complex filters (event correlation). This is, in fact, a general conjecture in system and protocol modeling research area.
- PN representation, in general, provides a better space complexity than the DFA representation [27].
- PN provides a better scalability with the number of events occurring in the system. In PN, event occurrence is represented by marking an event place with a token. Thus, events reserve only one place in PN, regardless of the number of occurrences. In contrast, in the DFA approach, every event occurrence is associated with one state in the DFA. This further implies that the number of states grows linearly with the

number of events occurrences. Therefore, in general, the PN approach may require less space complexity than the DFA approach in modeling event filtering.

However, the PN model is more complex to implement and manipulate than the DFA model. Appendix C.2 shows the data structure representation of PN as implemented in this architecture. PN is a list of nodes (PNnode) where each node consists of a list of states (places), expression (PNFX) and other node associated information such as the ID of this node (NodeID), the name of the represented filter (FilName), a flag to indicate if the node can be fired (fire_flag) and the actions to be performed (Action) if this PN node is fired (i.e., filter is triggered). Each place (Place) contains of a list of event states (OccurStates) such that each state represents an occurrence of this event. An event may occur multiple times with different attribute values. The *OccurStates* buffers all event occurrences information in FIFO order for later analysis. Users may decide to use the first, the last or all occurrences of the event instances for evaluating filter or event expressions (Expression). The *mode* member in *Place* class determines how event occurrences (OccurStates) must be handled (i.e., which event occurrence to consider). In addition, class *Place* contains the *mark* member which represents a PN *token* that indicates if this place is marked (see Section 2.2).

The expression (PNFX) in a PN node represents the *guard function* which is evaluated when all places are marked and the fire flag (fire_flag) becomes *true*. The PNFX is a list of predicates where each one contains a right and left attributes, and a binary relation (see predicate specifications in Section 3.3). In our architecture, PNFX represent the *event expression (EX)* and the *filter expression (FX)*.

The main interface of the `Iterator` classes for DAG and PN is illustrated in Appendix C. Finally, it is important to mention that LMAs and DMAs have an identical design architecture in terms of components, subcomponents and services and they only differ in the filter internal representation.

5.3.2 Subfilter Processor

When the subfilter delegation is received by MAs, the filter internal representation is updated to reflect the new monitoring operations. This process consists of two steps: (1) subfilter translation in which the subfilter specification is converted to a DAG or PN

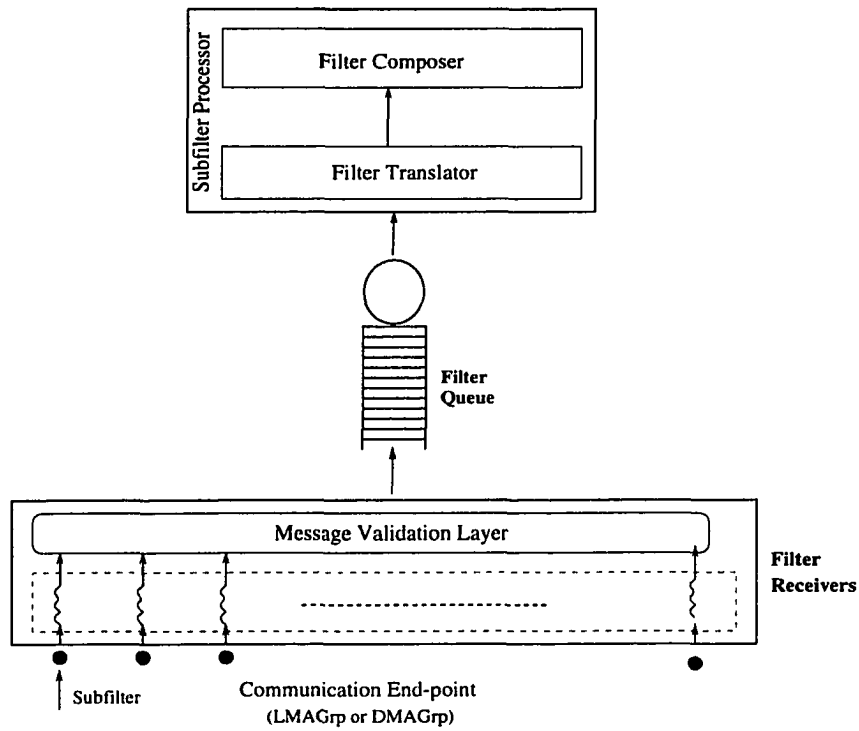


Fig. 5.5. Subfilter Processor Subcomponent.

representation (depending if it is an LMA or a DMA), and (2) filter composition in which the subfilter is inserted at the "proper" location in the internal representation (DAG or PN). The former is performed by the *filter translator*, and the latter is performed by the *filter composer*, as shown in Figure 5.5. This figure shows that when a subfilter is received via LMAGrp or DMAGrp, it gets validated to verify its destination and then it is buffered in FIFO order for the translation and composition processes.

5.3.3 Event Processor

The main function of the event processor is to filter incoming events based on the filtering internal representation such as DAG or PN. In this section, we discuss the implementation design, algorithms, and the optimization techniques used for developing this subcomponent.

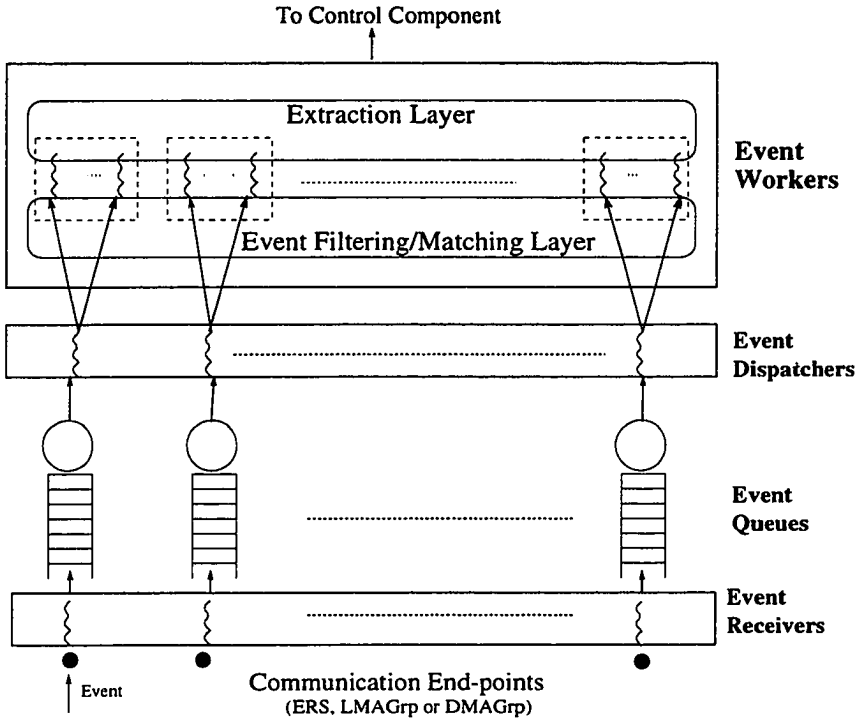


Fig. 5.6. Event Processor Subcomponent.

Subcomponent Implementation

The event processor is a major component in the architecture because it performs the most intensive process in the monitoring architecture. For this reason, the architecture of this subcomponent is deliberately designed to improve the throughput of the event filtering process and thereby improving the performance of the monitoring system. Figure 5.6 shows the internal architecture of the event processor subcomponent. The event processor is designed as a multi-layer and multi-threaded architecture. Each layer has a pool of threads that perform the function of this layer. This architecture enables different events from the same or different producers (ERSs or LMAs) to be processed concurrently and independently. Notice that this architecture is used in both LMAs and DMAs, therefore, the event producers in the first case are the ERSs, but in the later case are the LMAs associated with a DMA. The *event receivers* is a pool of threads where each one is connected

to different event producers (ERSs or LMAs) and working independently in receiving and queuing incoming events. After events are queued, they get dispatched from the *Event Queues* based on their priority by the *event dispatchers.* Then, the *event workers* perform the event filtering on the dispatched events based on the internal filtering representation (DAG or PN) constructed in MAs. If an event is detected, then the monitoring information is extracted by the *extracting layer* from the event notification and forwarded to the Control Component.

This architecture has the following design features:

1. *Providing the Maximum Degree of Concurrency:* The multi-threaded multi-layer provides *horizontal parallelism* where events from different sources (providers) can be processed (typically, queuing, dispatching or filtering) by threads in the same layer concurrently and the multi-layer provides *vertical parallelism* where events from the same source (provider) may be processed by threads in different layers concurrently. Although this parallelism can get the maximum advantage if the MA is working in a multiprocessor machine (e.g. sparc2000), using this architecture in a single processor machine has considerable advantages [52]: (1) it supports overlapped I/O (network or memory), (2) it allows overlapping computation and the blocking system calls (I/O) which increases the throughput and the performance of the monitoring system, and (3) it enables controlling the monitoring or the filtering process more effectively since a thread utilization of the system resources could be bounded by it priority. We will discuss this issue in more details below.

2. *Avoiding Threads Contention*: The multi-queue configuration is used in the event processing component to reduce the dependency between thread workers and thereby avoid the need for locking or mutual exclusion mechanisms which may decrease the system efficiency. This can be easily perceived in the case of event processor because of the large number of events that may be received in a short period of time. However, this is unlikely in the case of the subfilter processor or subscription component since the normal receiving rate of subfilters/messages is not high. For these reasons, the multi-queue configuration is used in the event processor but a global queue is used in the subscription component. The possibility of shared access and mutual exclusive requirements exist only between the event dispatchers and receivers. However, this

can also be reduced by limiting the locking (mutual exclusion) granularity on one queue slot rather than the entire Event Queue.

3. *Supporting Priority-based Monitoring:* The event processor architecture enables the developers/users to discriminate between events during the monitoring process based on their priorities. This is a significant feature for many monitoring applications such as fault recovery where events have different importance and priorities based on the failure type and component (see Section 7.2). The event processor supports the priority-based monitoring service via the following features: (1) *Priority-based Buffering:* events are dispatched from the event queue based on their priorities, and (2) *Priority-based Processing:* event workers which are Light Weight Processes (LWPs) can be assigned different priorities and they can also be suspended, scheduled and resumed [52] based on the priorities of the outstanding events. Thus, this feature not only enables discriminating between different events generated by one producer, but it also can distinguish between events generated by different producers by extending the event dispatching mechanism to consider a global priority scheme between event producers connected to an LMA.

4. *Portability to Multiprocessor Machine:* Using low-end multiprocessor machines such as Sparc2000 is feasible due to the drop in performance/cost value. The monitoring architecture can be ported to the multiprocessor environment to obtain the maximum advantage of parallelism with no extra effort required from the developers or the users. This is an important feature for monitoring critical applications such as military distributed interactive simulation where high-performance machines can be utilized to gain high monitoring performance and response.

Event Filtering Algorithm

In this section, we briefly describe the primitive and composite event matching algorithms in DAG and PN, respectively. Appendix C.3 and Appendix C.4 show the important part of the event matching algorithms in LMA (DAG) and DMA (PN). In *DAGEventMatch()*, upon receiving an event, the DAG is searched for common module names, and then common function names. If the event name is found, it then compares the event attributes again the attributes in the DAG (PredEvaluate). If all event attributes matches the DAG

attributes, the event name is returned, otherwise the "REJECTED" string is returned.

In *PNMatchEvent()*, the PN is searched for all places that may contain the received event. If a place matching this event is found and this place is not marked yet, then this place will be marked and the event information is stored. Also, the fire flag of this node is decremented indicating marking one place or an event occurrence. In our current implementation, we only consider the last occurrence of an event in the event/expression evaluation. When the fire flag becomes zero, the FX is evaluated and action is performed if the evaluation results is *true*. Otherwise, the filter places is restored for future evaluation.

5.3.4 Monitoring Optimization Techniques

In this section, we present various optimization techniques to reduce the computation and memory space required by the filtering process. This results in a considerable improvement in the performance and the scalability of the monitoring system.

- **Efficient Filtering of Primitive Events**

The filtering process involves comparing the information in event notification with the filtering representation information in MAs. As a result of the filtering process, the event is either "detected" if it matches an existing filter, or "rejected" if no match exists. This implies minimizing the Filtering Detection Time (FDT) and/or Filtering Rejection Time (FRT) which results in increasing the filtering throughput (number of events processed per time) and improving the monitoring performance. In this section, we propose two optimization techniques that reduce the time required to filter primitive events by minimizing both FDT and FRT:

Multi-path DAG vs. Binary-path DAG: The exiting event filtering mechanisms [12, 59, 63, 94] use a binary-path DAG (Direct Acyclic Graph) that includes two outgoing paths: true path and false path. The evaluation (i.e. comparison) result of a filter predicate determines the outgoing path that should be traversed. This DAG organization is inefficient because the `Attribute` in the filter predicate (see Table 3.2) could potentially have range of values. In this case, the number of predicate evaluations may grow linearly with the number of potential values in the attribute. For example, assume an `event type`

attribute that can take n different values. Using the binary-path DAG, in the worst case $O(\log n)$ comparisons are required to discover the outgoing path. However, the performance can be improved remarkably if a *multi-path* DAG is used, instead. In multi-path DAG, the value of the filter predicate attribute (`event type` in the example) is used as an access key to a hash table entry to obtain an index to the outgoing path. In contrast to a binary-path DAG, this involves a single predicate evaluation and a hash table lookup regardless of the attribute range length. This optimization technique reduces both FDT and FRT.

Parallel Filtering: Exploiting parallelism will obtain a significant increase in the performance of the event filtering. In our monitoring architecture (Figure 5.6), the event filtering mechanism is designed to provide the maximum degree of concurrency without congesting the filtering process with any locking operations (e.g. `mutex_lock` in Unix environment). Each event producer is assigned to a dedicated queue and each event is filtered by a separate thread or LWP (see Section 5.3.2). In addition to the ease of implementation obtained by using the multi-threaded architecture [52], this implementation can be utilized in the multiprocessor environment to provide a *parallel event filtering* mechanism. This will improve the filtering throughput by many orders of magnitude over the single processor.

- **Efficient Filter Composition**

The process of integrating one or more filters into the filtering internal representation during the subscription process is called the *filter composition* [3]. An efficient composition technique is important to produce an optimized filtering representation which reduces the filtering process time (FDT and FRT). As described in [3], different techniques have been proposed to achieve an efficient filter composition. These optimization techniques involve re-ordering the predicates of the filter internal representation in order to minimize the time required to process an event (i.e. FDT and RAT).

Matching Common Predicates First: When two or more filters are composed due to subscription process, the combined predicates are re-organized such that the common predicates of all composed filters will be matched/tested first before any other predicate.

This means that the common predicates are placed at the beginning of a filter internal representation. Assuming that the predicates re-ordering time is negligible, and evaluation time is almost the same for all predicates. As shown in [12], this technique minimizes the event filtering time by reducing the number of predicate evaluations of rejected events which do not satisfy at least one of the common predicates.

• **Space Requirements Optimization**

In this section, we describe the problem of the growing demand of memory that may be required in the DMAs as a result of detecting composite events. DMAs must store the event history in order to evaluate the filter expression. For this reason, the excessive space required in the DMA due to the continuous growth of the event history is a major concern in our monitoring system design. In this section, we proposed different techniques to control space requirement and avoid space explosion in DMAs:

Distributed Event History: The monitoring is a hierarchical distributed architecture which naturally distributes the event history among the DMAs. Although this distribution reduces the impact of the event history growth, it may not be sufficient because (1) the event distribution may not be *balanced* since some DMAs could be exposed to more events than the others, or (2) it may not be *enough* since the global event history could be more than global space provided by all DMAs. For these reasons, we present in this section some other techniques to optimize the space requirement in the DMAs. However, our impression from using the monitoring system in different application environments (see Section 7.2) is that relying on this feature is sometimes sufficient. This could be due to the nature of the monitoring applications in which a limited number of filters can coexist simultaneously, and also the producers which generate a symmetric event distribution in the monitoring domains.

Using Event Timeout: This feature enables the users or the developers to specify a timeout period for the composite events components (i.e. primitive events). When an event timeout expires, the event-related predicates are revoked from the internal representation. The event timeout, *Timeout* can be either *absolute* where the event predicates are eliminated *Timeout* (e.g. 5 seconds) after the they are inserted, or *relative* where the event

predicates are eliminated *Timeout* after the occurrence of another event. This technique is especially useful when the temporal relation between the events is predictable. For example, assuming that the event "receiving a message of a sequence number x" or R_X is not detected for sometime period. Then, the predicates of the event R_x are reclaimed when the event "receiving a message of a sequence y" or R_y such that $y > x$ is detected*. Thus, in other words, $Timout_{R_x} = DetectTime_{R_y}$ which means R_x event expires when the event R_y is detected. This technique is useful to place an upper bound on some events which are likely to occur during a certain point of time (relative or absolute). For example, in reliable networks, time period between sending a message, S_m, and receiving the same messages, R_m, is certainly bounded. Thus, we can say $Timeout_{R_m} = DetectTime_{S_m} + MTD$ such that time MTD is the maximum transmission delay including the retransmission time. This event timeout technique assists in reducing the space requirements by continuously eliminating event predicates that belonging to an expired event (e.g., event that will never occur). It is the users' responsibility to determine and specify the event timeout using HFSL.

Filters/Subfilters Delegating: In some environments, predicting the events timeout may not be feasible. Therefore, this technique reduces the space requirement by enabling the DMA to forward the assigned filters' (or subfilters') internal representation to another DMA which in turn has a sufficient space to handle the new filtering responsibilities. Each DMA sent this "delegation" request to its higher DMA in the hierarchy until one upper DMA accepts the request or a new DMA is allocated using the dynamic hierarchy feature described in Section 4.2 for handling this excessive demand. If the delegation request is granted by a DMA, the original DMA requests forwarding all events belong to the delegated filters to the new DMA. Although this approach utilizes distributed events buffering in MAs to minimize the space requirement, it incurs a communication overhead due to the process of the MA delegation protocol.

*Assuming the sequence number do not circulate.

5.4 Control Component

The control component is provided to support reactive control monitoring applications. The main function of this component is to perform the actions specified in a filter program. There are four types of actions supported by the monitoring architecture: program execution, information dissemination, event generation and filter incarnation. Section 3.3 discusses the formal specification and the applications of these four actions. As shown in Figure 5.1, the control component has two major subcomponents:

Dissemination Service

It is used to forward the monitoring information to corresponding consumers based on the subscription criteria. There are two important issues to consider in this component: (1) information may be disseminated to groups of consumers which implies employing a reliable multi-point (multicast) communication protocol in order to achieve an efficient service, and (2) unlike traditional multicast groups, the consumer multicast groups change dynamically based on their subscriptions (filters). However, the traditional multicast protocols forward the information based on the IP address and port number only. In other words, the dissemination process considers which consumer subscribes to which filter rather than multicast group/address. One naive approach is to make all consumers join one multicast group such that monitoring information is sent to this group and it is the responsibility of the consumer program to consider or ignore messages based on users' subscriptions. Although this solution is simple, it incurs considerable overhead on the consumers and may generate a large network traffic. A second approach for solving this problem is to assign a unique multicast group for each filter in the system. Therefore, consumers subscribed to this filter will receive the monitoring information via the multicast address corresponding to this filter. However, this may consume a large number multicast addresses and system resources (e.g., file descriptors) since consumers may have to join and listen to a large number of multicast groups. So this solution is not scalable and complicates the consumers' tasks. Our solution uses the *dynamic group masking* scheme proposed in [4]. In this scheme, each consumer is assigned a unique identifier (CID) in the monitoring environment. The monitoring agents include the CID(s) of one or more consumers in the monitoring information messages which are then is used by RMS

to decide to ignore or forward this message to its consumers. This solution relieves the consumers from the overhead imposed by the former approach, however, it does not reduce the resulting network traffic. This requires a solution in the router level similar to the one proposed in [65]. Discussing such approach is beyond the scope of this dissertation and left for the future work (see Section 9.5).

The monitoring agents use this subcomponent when a `FORWARD` action is found in the filter program or when sending control and status information to the corresponding consumers. The dissemination services is also used for group communication between the monitoring agents themselves using RMS [4].

Action Service

The Action service subcomponent is used to (1) execute a local program, (2) send a request to execute a remote program, (3) generate a specified event, or (4) send requests for adding, deleting or modifying filter programs. Execution of a program is performed via `fork` and `exec` system calls by the agent that detects the event pattern. If executing a remote program that resides on hosts different host than the agent's host is desired, then consumers can achieve this by defining an event (request) to be sent to this "remote service" which executes the target program. This remote service is similar to the user action service depicted in Figure 5.4. The action service also sends a new specified event or a request of filter manipulation to LMAGrp and/or DMAGrp.

5.5 Adaptive Object-Oriented Filtering Framework

The goal of this section is to describe the object-oriented design and implementation of an adaptive event filtering framework which can be integrated and reused efficiently to develop event management applications for various domain environments. In our approach, the event filtering framework captures the *common components* and *design patterns* of event management in different domains. The major contribution of this work is to provide a flexible event filtering framework that can be efficiently adapted to different domain-specific requirements and with minimal development effort. HiFi monitoring system is an example of using this filtering framework in distributed system and network management. We also present an example of using the filtering framework for developing event

management applications in a different domain.

5.5.1 Motivation

The significance and the broad deployment of the event filtering in several application domains is the primary motivation for developing an Object-Oriented filtering framework that encompasses the common components and design patterns required by event management applications. The framework enables developing customized filtering mechanisms that possess different alternative specifications based on the domain requirements and characteristics. This is obtained by facilitating the reuse of the code, design patterns and the fine-grain design modularity of the framework components which produce an adaptive filtering framework for various event management application domains. This framework improves the reliability and performance of event management applications while minimizing the development effort and cost.

5.5.2 Event Filtering Framework Components

The Object-oriented application framework is a reusable, semi-complete application that can be utilized to produce a custom applications [42]. In this section, we describe the object-oriented components of the event filtering framework that support the basic infrastructure and services required in several application domains. Developers in different domains can integrate, reuse and extend the framework components to develop domain-custom event management applications. Figure 5.7 shows the object diagram of the filtering framework.

Event Definition Constructor Component: Event management applications require the users to specify the events format prior to the filtering process. The Event Definition Constructor (EDC) component is a set of related objects that provide basic interfaces to define events. In order to emphasize the design modularity, this component is divided into several classes that construct *event attributes, event operators (basic and advanced), primitive events* and *composite events* which represent the basic elements of event definition (called hot spots [76]). Developers can directly reuse the services provided by the EDC interfaces via object composition or they may customize this component by developing dif-

ferent alternative specifications of the event elements. For example, developers can define different event operators or composite events models. This feature is important to make the event filtering framework extensible and adaptable to various application domains that have variations of event elements definitions.

Subscription Component: The subscription component uses the *filter programming interface (FPI)* (see 5.7) class in order to validate and interpret the filter programs defined by the users (interpreter pattern). The second task of the subscription component is to build the filter itself by constructing the filter internal representation which is a connected graph representation conveying all information peculiar to this filter. The subscription component uses the *filter builder* class to achieve this task. Therefore, the subscription component uses these two classes to separate the filter construction from its actual representation (builder pattern) which provides a broad adaptation in the filtering framework. For example, a filter represented in any programming interface can be constructed using many different filter internal representation such as DAG or PN and vice versa. This is a significant feature for event management systems since applications domains use various models of filter programs (i.e., programming interface) and number of different internal representations such as DAG and PN depending on the domain requirements.

Filter Iterator Component: This component is also called a filter/event processor component. The main task of this component is to operate the filter internal representation constructed by the subscription component as described above. In other words, it represents a set of algorithms used to access and manipulate the elements of the filter internals (e.g., DAG, PN or DFA) without exposing its underlying representation (iterator pattern) [26]. Developers can reuse the iterator algorithms in the *filter composer* class to insert, delete and modify filter programs in the internal representation. The filter iterator component also has the *event processor* class which inspects incoming events from the observed system and determine if an event is *detected* or *rejected.* The filter composer permits the developers to customize the framework to adopt different alternatives filter internal representations such as DAG and PN. In addition, various filtering optimization techniques can be applied using the event processor class independently from the filter internal representation itself. This makes the filtering framework adaptable to many al-

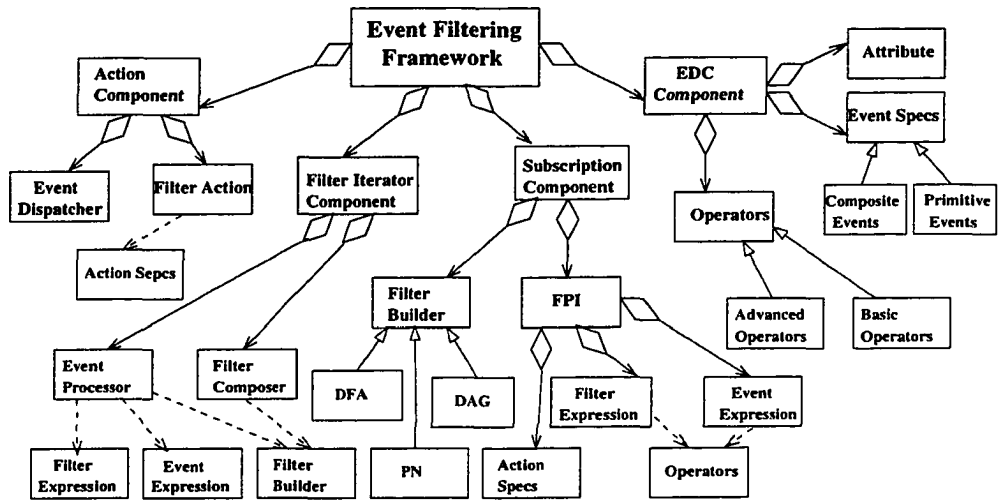


Fig. 5.7. The Event Filtering Framework Classes.

ternative design issues in event management applications.

Action Component: Whenever an event of interest is discovered, the filter iterator component instantaneously notifies the action component which consequently performs the action(s) specified in the filter program. The action component classes use the information provided by the subscription component (i.e., filter constructor) to identify the corresponding action of a specific filter. The action component classes can be easily customized to perform general or specialized actions related to the domain environment. For example, an action can be the invoking of methods, executing programs or dissemination events to corresponding management applications. In Section 7.2.4, we presented management action examples for supporting fault recovery in distributed systems. The action component also provides an events dispatching mechanism, via the *event dispatcher* class (see 5.7), based on input/output (I/O) functions or timers routines.

5.5.3 Event Filtering Framework Applications

Event filtering can be used to manage systems in real-time by tracking and classifying applications events. In this section, we briefly describe how the event filtering framework

can be incorporated within a common utility such as Electronic Mail (EMail) to provide an efficient management of the received messages.

Electronic mail (or EMail) is a very commonly used application and people like to have an efficient EMail management tools for their mail events. The event filtering framework can be easily incorporated with existing EMail systems to provide a dynamic control for incoming mails. For example, users can instruct the Email event management system which incorporates the filtering framework to discriminate between incoming mail messages based on EMail event attributes such as `Sender`, `Subject`, `Status` or even specific words in the message text. Based on the user interest expressed in the action component, the EMail messages can be, for example, ignored, forwarded to other devices (such as digital pagers or printers), categorized/sorted based on their priority or even disseminated to groups or users. Similarly, users can trigger actions based on a correlation of two or more EMail events.

5.6 Summary

The HiFi monitoring system consists of the following four major components: *Instrumentation* used for inserting monitoring sensors in the application code, *Subscription Service* used for processing the monitoring information (events, filters, environment) and in turn performing the filter decomposition and allocation task, *Event Filtering* that constructs the internal filtering information, and inspects received events according to filter subscriptions, and *Control Component* which is used for performing the filters actions if the event correlation is detected. The first two components constitute the manager or the event consumer program, while the last two components constitute the monitoring agent architecture.

The instrumentation components automatically replaces the "user sensors" include the event name only with extended "system sensors" that convey the all information needed to construct event notifications. In addition to this automatic sensors insertion, the instrumentation component supports *dynamic signalling* to enable activating and deactivation reporting events dynamically, and *adjustable event reporting* to specify the delivery or reporting time of an event (immediate or delayed reporting). The subscription components parses the monitoring information and construct the monitoring-base knowledge

(MB). Based on MB information, agent are organized (using the automatic agent organization protocol described in Section 4.3.3) and filters are decomposed and allocated.

The event filtering components represent the core of the monitoring agent. This components uses DAG representation for the LMA but PN for DMA in order to track events history. The subfilter processor in this component inserts the received monitoring delegation at proper places in the DAG or PN (filter decomposition technique). However, the event processor subcomponent parses the received events and matches them according to the users' subscriptions exist in the DAG and PN. This chapter presents number of optimization techniques that reduce the time required for matching an event in the DAG such as exploiting the parallel filtering offered by the multi-threaded and multi-layer architecture of the monitoring agent, and matching the "common predicates" first in the event filtering process. Other techniques are presented to minimize the space required by the PN such as associating event timeout in the filter specification. When a monitoring agent detects the specified event correlation, it invokes its action component to perform the associated action. Actions types are described in detail in Chapter 2.

CHAPTER VI

PERFORMANCE EVALUATION

Throughout the survey of literature in monitoring distributed systems area, only a few papers were found that numerically reported monitoring system performance. Most of these focused on system perturbation, [55, 58, 69], or are only valid for LSD systems with shared memory [32]. Additionally, none addressed or evaluated the issue of monitoring system scalability. This Chapter describes a performance evaluation study of the HiFi monitoring system. To conduct this study, benchmarking routines and simulation programs were processed in a benchmark/simulation testbed environment so that application *perturbation* (e.g., intrusiveness), *scalability* and *throughput/latency* could be assessed. Results from this study are presented numerically. An overview of the testbed environment, workload characterization, and benchmarking routines is also provided.

6.1 Workload Characterization

In order to develop a workload that can be used repeatedly, we use a *synthetic workload* whose characteristics is similar to those in real workload but it offers a flexibility of repeating and modifying the experiments without changing the system operation or handling large trace files [41]. In some cases, *random event generator (REG)* programs are used to emulate the applications events under certain controlled rate and distribution. The REG emulation programs are used for measuring the perturbation and throughput/latency of the monitoring system. On the other hand, simulation routines based on the HiFi monitoring system model described below are used for evaluating the scalability of the system.

Selecting the *workload parameters* is very crucial because it determines the validity of the resulting workload model. In this study, we identify the major workload parameters that effect the perturbation, scalability and throughout/latency of our monitoring system.

The workload parameters considered in this study are:

- *Event length*- Number of attributes included in a generated event impact the monitoring performance since each attribute could represent a comparison operation in the filtering algorithms. Event attributes are mixed of integer, floating point and string attributes and represent an average of 4 bytes per attribute.

- *Events rate and distribution*- It is also called event generation frequency which indicates the number of events generated per unit time. Another representation of this workload parameter is the *event generation probability (EGP)* which is the event generation likelihood during the program execution. For example, EGP is 0.4 means 40% of the program instructions generate events on average. The *Bernoulli distribution (BD)* is used for generating random events. The BD is used to model the probability of an outcome having a desired class or characteristics such as a packet in a computer network reaches or does not reach the destination [41]. And this is typically the case in our applications since each instruction generates or does not generate an event. In addition, the BD memoryless property implies that trials are independent and the probability of generating an event is not affected by outcomes of the past trials. This property is important and used in characterizing many problems in computer networks such as network traffic [25].

- *Number of event producers*- This indicates the number of active application entities that are concurrently engaged in the monitoring process.

Since it is not the goal of this research dissertation to produce a comprehensive performance evaluation model for distributed monitoring systems, only workload parameters that we experimentally found have a major impact on the monitoring process are considered in the evaluation model. It is sufficient in our case to show the impact of our architecture and enhancements techniques on improving the scalability and performance, and minimizing the intrusiveness of the monitoring system.

6.1.1 Perturbation Analysis

In this section, we describe the experiments and the results of benchmarking and evaluating the perturbation of HiFi monitoring system. The time measurements in these experiments

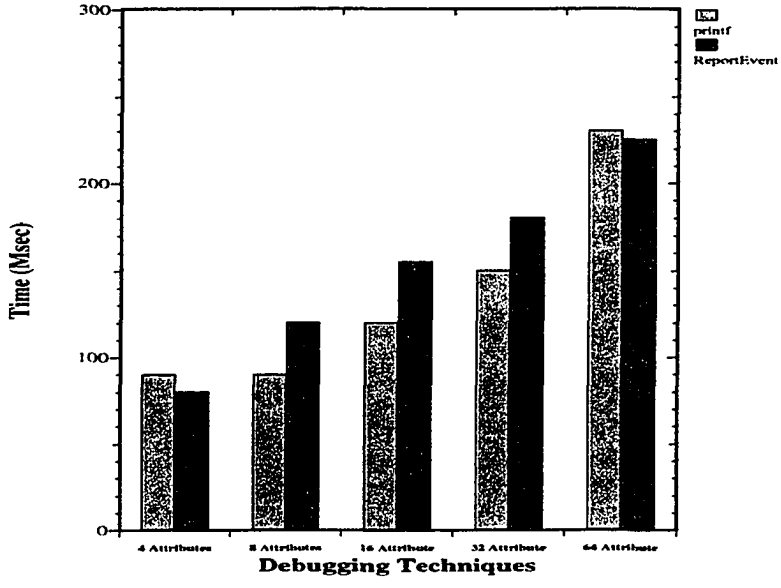


Fig. 6.1. ERS ReportEvent Perturbation.

are performed using the standard UNIX high precision timer routines [84].

The application perturbation can be measured by the execution time overhead caused by the monitoring operations including the ERS event reporting process, and monitoring agents (LMAs and DMAs) operations.

ERS Effect

In order to show the effect of the `EventReport()` function call, we compare its overhead with the traditional C `printf()` function which is frequently used by programmers as a simple way for debugging and inspecting the program state and behavior. Figure 6.1 shows that the overhead of the `EventReport()` is comparable with `printf()` and ranges between 100 to 200 microseconds based on the event length. Similar results are found when using other printing functions such as `cout` in C++. This implies that the event signaling (generation) process performed by ERS which includes function transfer, event construction, and event sending causes minimal overhead and could be neglected as a

turbation factor in our analysis.

Application Perturbation Measurements

In this experiment, we measure the actual overhead caused by the monitoring system including reporting time (ERS processing), primitive event filtering time (LMAs Processing), event correlation time (DMAs Processing), UNIX socket communication and the RMS communication. In order to measure this experimentally, we use the filter correlation example, *HelloWorld* filter, described in Appendix A. In this example, two REG emulation processes located in different hosts in the same LAN generate up to 5000 events randomly using Bernoulli distribution. At each event time, REF process may choose to generate or not to generate *Hello* or *World* events with a probability of 0.5. Each REG program is connected to an LMA residing in the same machine. When events are received by an LMA, the LMA checks the event and forwards it to the DMA located on a different machine than the LMAs but on the same LAN. The DMA send a notification to the manager if it receives *Hello* event and *World* event of the same sequence number (*TStamp*) from two different REG processes (i.e., machines) :

$$HelloEvent.TStamp = WorldEvent.TStamp \wedge$$
$$HelloEvent.Machine \neq WorldEvent.Machine$$

In these experiments, the REG programs are fist run without instrumentation or monitoring. Then the programs are instrumented and run in HiFi environment to detect the event of interest. Figure 6.2 depicts the results of this experiment with various event generation probabilities (EGP). Figure 6.2 shows that the perturbation increases when EGP increases. However, this increase in perturbation may be considered low compared with other monitoring systems such as Issos [69] (61%) and Falcon [32] (> 40%) when the event rate in between 10% to 30% of program execution. Furthermore, the perturbation increase is not linear with EGP, which indicates a slower growth with event rate. Practically, typical programs in distributed or nondistributed environment are unlikely to generate higher than 20% which represents using the `EventReport()` (or generating an event) every four instructions in the program. All generated events are 8 attributes long. Machines used in this experiments are Sun Sparc 5 with Solaris 2.5 connected with Ethernet of 10 Mbps. The emulation code for ERG programs is listed in Appendix D.

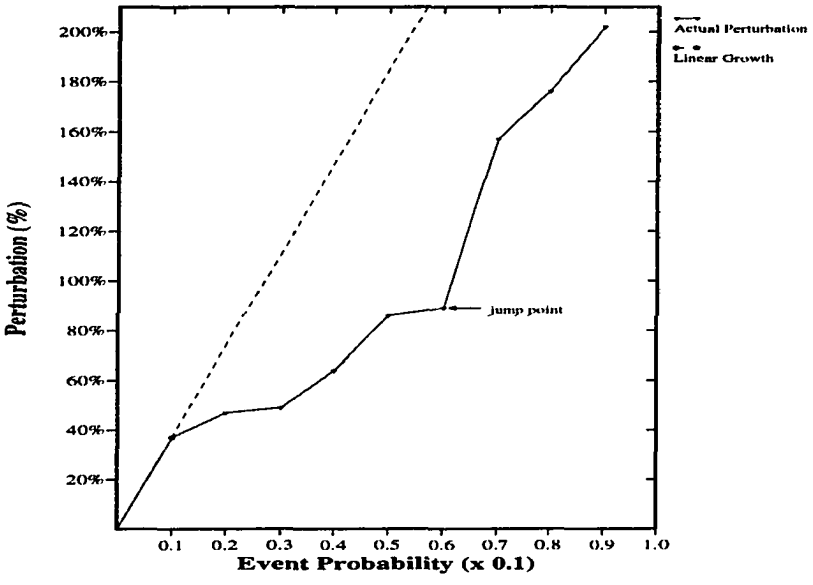


Fig. 6.2. Application Perturbation.

Minimizing Perturbation

Several techniques are developed to minimize the application perturbation including *dynamic signaling* and *events batching* which are described in Section 5.1.3 and Section 5.1.4, respectively. To measure the effect of dynamic signaling, the REG programs were changed such that 50% of the generated events are filtered out by ERS. To measure the impact of both dynamic signaling and event batching with maximum of 5 events, REG and ERS programs has been changed to reflect this effect. Figure 6.3 shows a substantial improvement in reducing the application perturbation. As described Section 6.1.3, the perturbation is mostly impacted by the communication overhead resulting from agents and application interaction.

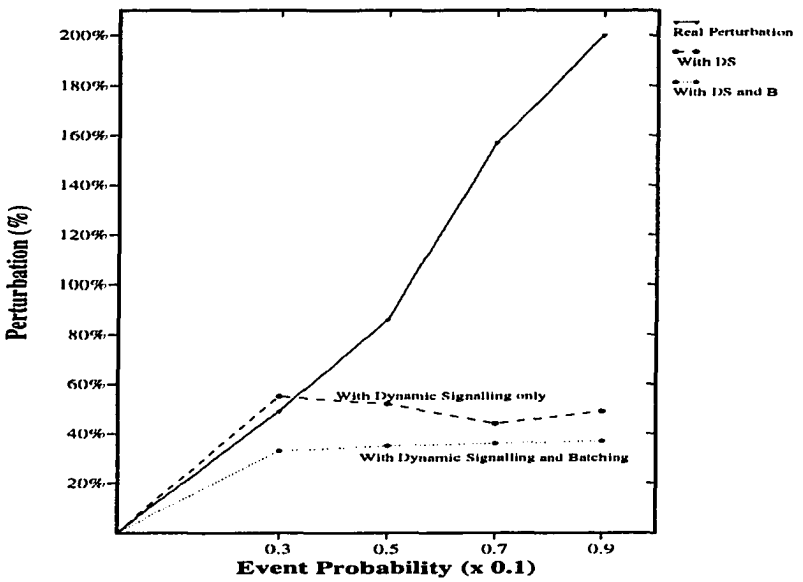


Fig. 6.3. Minimizing Application Perturbation.

6.1.2 Scalability

Our approach for evaluating the scalability of HiFi is to measure the impact of increasing the event frequency and number of event producers on the mean response time or the monitoring latency. The monitoring latency is the elapsed time between the event occurrence and the manager notification. In other words, it is the event processing time by all monitoring entities. We also compare the mean response time (monitoring latency) of the hierarchical filtering approach with the centralized and decentralized monitoring approaches described in Section 4.1. In order to perform this test, we developed simulation routines to simulate each of these monitoring approaches using M/M/1 model technique [41]. The simulation routines assume that (1) event arrivals (λ) are exponentially distributed, (2) the average monitoring/filtering service rate (μ) of an agent is 8000 events per unit time (second) which was experimentally derived from Figure 6.7 discussed in Section 6.1.3, (3) 50% of the events are interesting (i.e., the probability of an event

to be accepted is 0.5), (4) for the sake of simplicity and since we are in a comparison study, the communication overhead, event generation time are neglected because they represent constant overhead in each approach and do not add a value to our comparison study. However, as we will show latter in Section 6.1.3, the actual monitoring latency as experimentally measured including the communication overhead. The mean response time (MRT) is given as follows [41]:

$MRT : E[r] = (1/\mu)/(1-\rho)$ such that $\rho = \lambda/\mu$

In the following, we briefly describe the simulation models and routines.

Centralized Monitoring Simulation: In a centralized monitoring architecture, event producers send their generated events directly to a centralized monitoring node, where event filtering and correlation are performed. This architecture can be represented in M/M/1 model such that $\lambda = f * N$ where f is the event frequency and N the number of event producers:

$$MRT_{centralized} = (1/\mu)/(1-(f*N)/\mu)$$

Decentralized Monitoring Simulation: This architecture is similar to the previous one except that event filtering is performed by a local agent in the in the producer host before forwarding it to a central node for event correlation process. Therefor, there are two levels of processing/filtering: (1) by the producer agent, and (2) by the centralized monitoring node. Thus,

$$MRT_{decentralized} = (1/\mu)/(1-(f/\mu)) + (1/\mu)/(1-((f*0.5*N)/\mu))$$

Hierarchical Monitoring Simulation: In HiFi, the monitoring latency is typically the event generation time (EGT) and the agent filtering time. However, the agent filtering time includes the LMA filtering time (LFT), the DMA(s) filtering time (DFT) in the monitoring hierarchy, and the agent communication overhead (C). Based on this workload characterization, we can define the monitoring latency as follows:

$$Latecny_{hierarchical} = EGT + LFT + \sum_{i=1}^{h-1}(DFT_i + C)$$

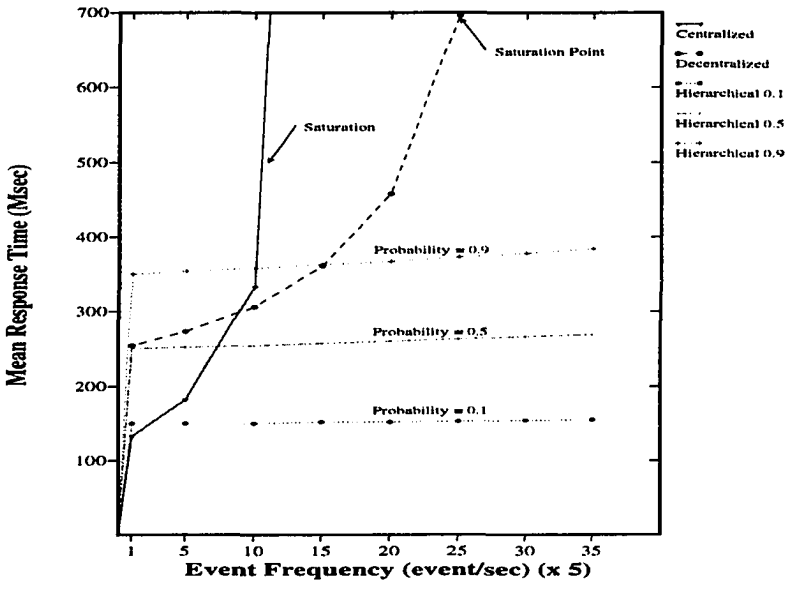


Fig. 6.4. Monitoring Scalability with Event Frequency.

where the hierarchy height: $h = \lceil log_x(N) \rceil$ where x is the branching factor and N is the number of producers.

However, as described before, EGT and C are neglected in the simulation model. Thus, the mean response time can be expressed as follows:

$$MRT_{hierarchical} = (1/\mu)/(1 - ((0.5 * f)/M)) + \sum_{i=1}^{h-1}((1/\mu_i)/(1 - ((f * 0.5 * x)/\mu_i)))$$

The first factor represents the LMA filtering and the second represent the DMA filtering. Notice that the LMA receives only $0.5 * f$ of the events since 50% of such events on average are filtered by ERS. Since not every event is necessarily forwarded up all the way in the DMA hierarchy, we calculate the MRT considering three different probabilities for forwarding an event: $10\%, 50\%$ and 90%. We also assume that there are a maximum of 10 LMAs are connected to one DMA in any domain ($x = 10$). We found this a valid

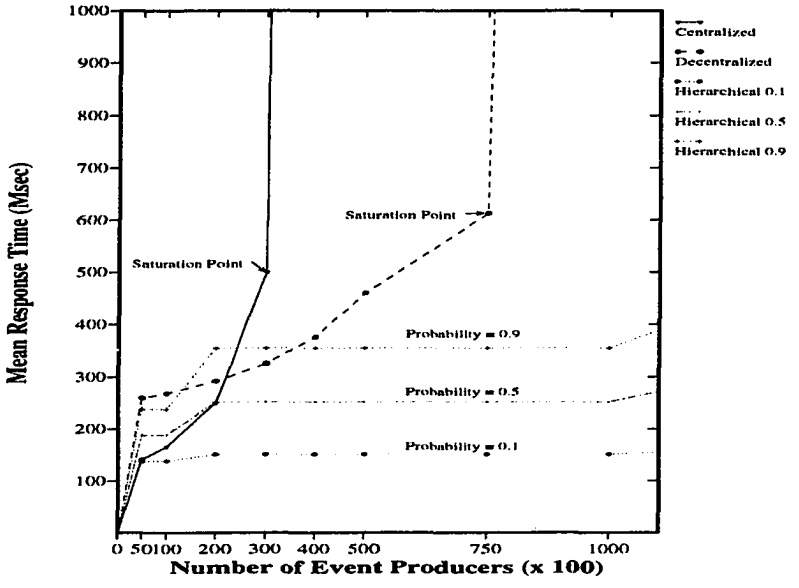


Fig. 6.5. Monitoring Scalability with Number of Event Producers.

practical assumption to be used in such experiment.

Scalability Simulation Results

The simulation results of monitoring mean response time versus event frequency and number of producers of the three approaches are depicted in Figure 6.4 and Figure 6.5, respectively. In the first figure, N (number of event producers) is considered to be 500 and in the second one f (event frequency) is considered 20 events per second. Both figures show the superiority of the response time (latency) of the hierarchical architecture over the centralized and decentralized ones. The saturation points in the figures indicates a buffer overflow and indefinite response time since $\rho > 1$ in this case. The hierarchical architecture with probability 0.9 is still superior over the other architecture because of the use of dynamic signaling. These figures also show that the MRT of hierarchical architecture grow slowly with respect of event frequency and number of producers. In fact, the "jump points" in hierarchical graphs in Figure 6.5 represent creating a new level in the hierarchy

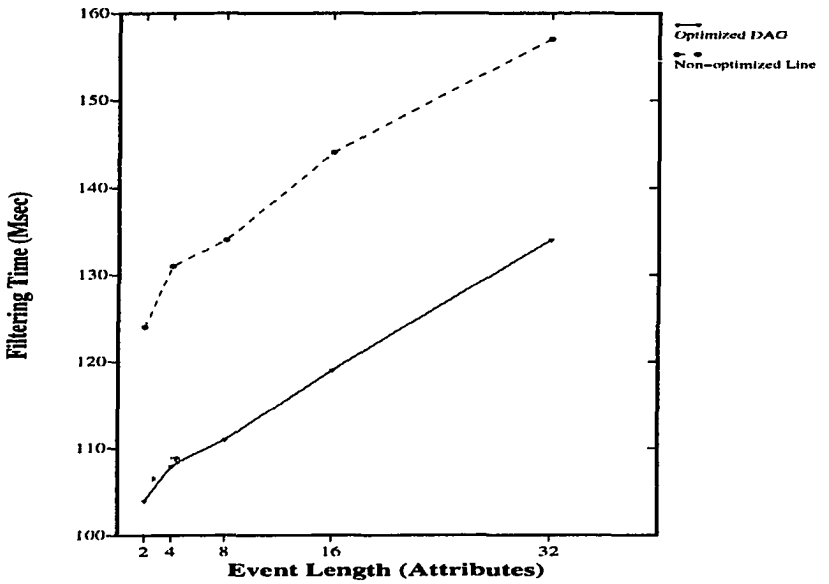


Fig. 6.6. LMA Filtering Latency.

to accommodate additional producers. These figures also show that the centralized and decentralized approaches have a better MRT than the hierarchical when event frequency is low and very small number of event producers exist in the system.

The simulation programs that implement such models are presented in Appendix D.

6.1.3 Throughput/Latency

This section presents a benchmarking results for the throughput of the filtering mechanism, `DAGMatchEvent()`, performed by an LMA. The event filtering throughput means the number of events processed per unit of time. Figure 6.6 shows the improvement of the optimized DAG algorithm over the non-optimized which is about 20% increase. The optimized DAG uses hash table lookup instead of binary branching DAG. This optimization mechanism is described in Section 5.3.4. Figure 6.7 is depicting the same data in the Figure 6.6 to show the number of events that can potentially be processed (i.e., filtered)

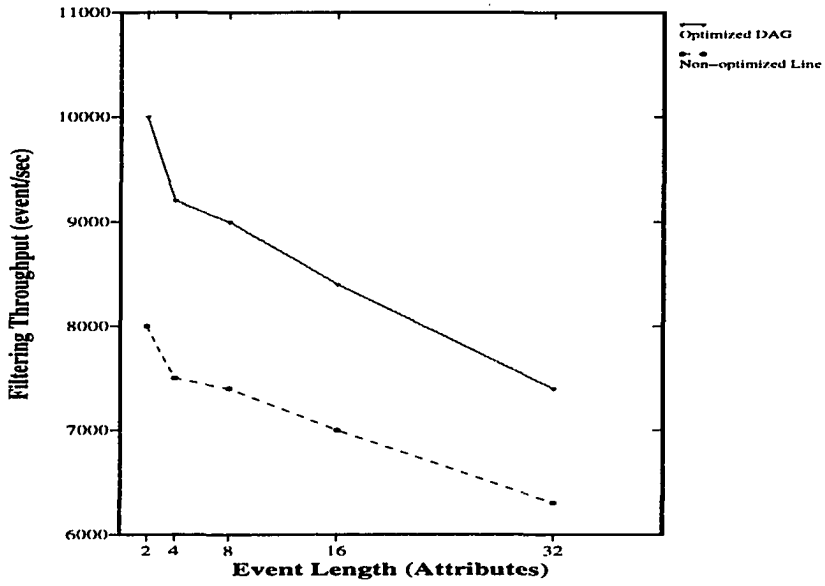


Fig. 6.7. LMA Filtering Throughput.

by one LMA. Form this figure, we choose μ to be 8000 as in the previous simulations. It is important to notice that these experiments measure only the filtering throughput isolating the UNIX and RMS communications overhead. The timer starts and stops before and after the event filtering function `DAGMatchEvent()` in the LMA code.

The next Figure 6.8 shows the actual monitoring latency as measured using the REG emulation programs and *HelloWrold* filter with different event rates. In this experiment, events are always generated (event generation probability is 1) until the maximum number of events is reached. We repeated the experiment for different maximum number of events: 1000, 2000, 3000, 4000 and 5000 events, and measured the average latency of each detected event correlation.

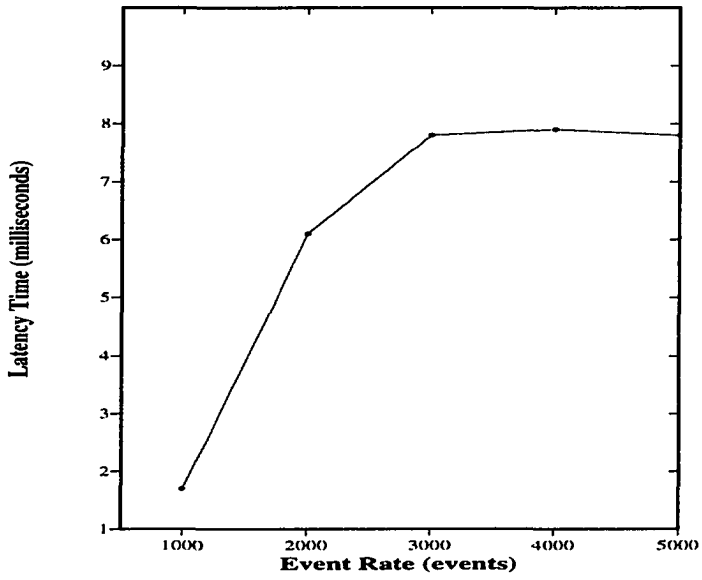


Fig. 6.8. Monitoring Latency.

6.2 Summary

To adequately evaluate perturbation, scalability, and monitoring system throughput, a number of benchmarking and simulation tests were conducted. As part of this evaluation, techniques to optimize the monitoring system (e.g., enhance monitoring operations or reduce its intrusiveness) were identified. Results show that the dynamic signaling and batching techniques used within HiFi significantly reduce application perturbation. ERS was also shown to have a minimal effect on application perturbation, which is important to the production of efficient instrumentation routines. The average monitoring intrusiveness of HiFi is considerably lower than that reported by other monitoring systems, such as Issos [69] and [32]. Throughput benchmarking demonstrated the viability of the DAG optimization technique. This technique resulted in a 20% improvement over the traditional DAG matching algorithm used in event filtering [59, 63, 94]. Simulation testing demonstrated that the hierarchical filtering approach is scalable in terms of event

frequency and the number of event producers in a centralized or decentralized environment. Finally, it is important to note that UNIX and RMS communication primitives are the primary source of overhead in the monitoring operation. This is explained by the high latency figures (in order of milliseconds) depicted in Figure 6.8. Regardless, filtering throughput is relatively high (in order of hundreds of milliseconds) and event generation time is negligible (in microseconds). This analysis factored out the effects of agent filtering and of the `EventReport()`, which implies that communication overhead had the largest impact on overall monitoring latency and throughput. Other monitoring systems exhibit this same characteristic and share this observation [58].

CHAPTER VII

APPLICATION EXAMPLES

Our monitoring approach decouples management tasks from application implementation, which offers significant flexibility in the application environment. Procedures, both monitoring and management, can be simply modified and are able to be replaced independent of the application implementation. Consumers and managers are free to experiment with different monitoring demands without restarting the monitoring system or the application. The following presents the case study of using the HiFi monitoring system with the Interactive Remote Instruction (IRI*) system. IRI is a collaborative distributed multimedia system that was developed at the Old Dominion University (ODU) to support distance learning [56]. The goal of this study is to illustrate the usefulness and effectiveness of employing HiFi for managing the complexity of a large-scale distributed system such as IRI. Examples of various monitoring applications using HiFi are also presented.

7.1 Case Study: Monitoring IRI System

The major software components of IRI are Session Control (SC) and Reliable Multicast Service (RMS), which are used respectively for resource management and group communication. IRI provides full interaction via the following components: Audio, Video, Presentation Tool (PT) and Tool Sharing Engine (TSE) components. Figure 7.1 shows the IRI interface in a typical classroom session.

IRI is a large-scale distributed multimedia system that may involve large numbers of users, application entities, and associated interactions (e.g. hundreds of students and processes). IRI is also typically implemented with a wide geographical distribution.

*http://www.cs.odu.edu/~tele/iri/

This implementation make IRI more susceptible to reliability and performance problems (e.g., component failures, errors, performance bottlenecks). Monitoring IRI components at run-time is essential to providing feedback information on IRI behavior used to improve reliability and performance of the IRI session.

This classroom includes all sites that are participating in any given IRI session. The ODU has been using IRI, still considered to be a prototype system, to conduct computer science courses during the last three semesters. At ODU, IRI connects three remote sites located about 200, 45 and 15 miles away from ODU main campus to the Computer Science department in the main campus to create a single virtual classroom [56]. This provided practical opportunities to monitor and support run-time reliability and performance issues [9].

This section describes the monitoring architecture used in the IRI sessions and then presents some of the monitoring applications which significantly increase the reliability and performance of the IRI system.

7.1.1 Monitoring Architecture in IRI System

In order to monitor the IRI system, IRI components (such as RMS, SC, Audio and Video) must first be instrumented so that all errors, warnings or important status information messages will be passed to the ERS. This means redirecting these messages from their traditional, standard output locations.

Based on these messages and IRI event specifications, ERS constructs the appropriate event notifications and sends them to its LMA. Even though the program may include many instrumentation instructions, the ERS is invoked when events correspond with existing subscription demands. This is achieved through global variables that are shared between ERS and the application itself. These variables are manipulated by the subscription component using ERS criteria (see Figure 5.1).

IRI events are divided into different categories, which are hierarchical in nature. For example, the *Failure Events* category is divided into `Major_Errors`, `Minor_Errors`, `Hard_Warnings` and `Soft_Warnings` subgroups. IRI event specifications are defined by the developers according to the HESL. Based on the monitoring model terminology (see Section 3.1), IRI software entities are classified and described as follows:

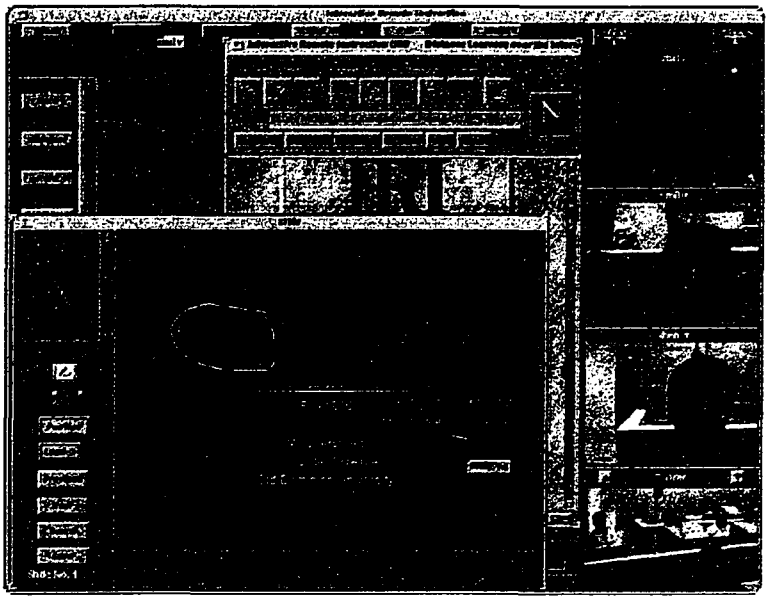


Fig. 7.1. Interactive Remote Instruction (IRI) System.

1. *IRIManager* is an event consumer program used by IRI administrators to monitor the system at run-time;
2. *IRI tools* are event producers (e.g., Audio, Video, PT and TSE) that emit events during their execution; and
3. *IRI modules* are typically the SC and the RMS components that act as both event producers and event consumers simultaneously.

As event producers, IRI modules generate event notifications to express their run-time status including communication failure events. As event consumers, IRI modules may request global feedback information about the session from other entities. For example, since the audio packets in IRI are sent unreliably, the SC of the sender may frequently request the average drop rate in the audio stream. Similarly, the RMS of the sender may request some information from other RMS(s) in order to discover slow members in the multicast group (to be discussed later in Section 7.2.3).

As shown in Figure 7.2, an LMA is assigned for each workstation in IRI environ-

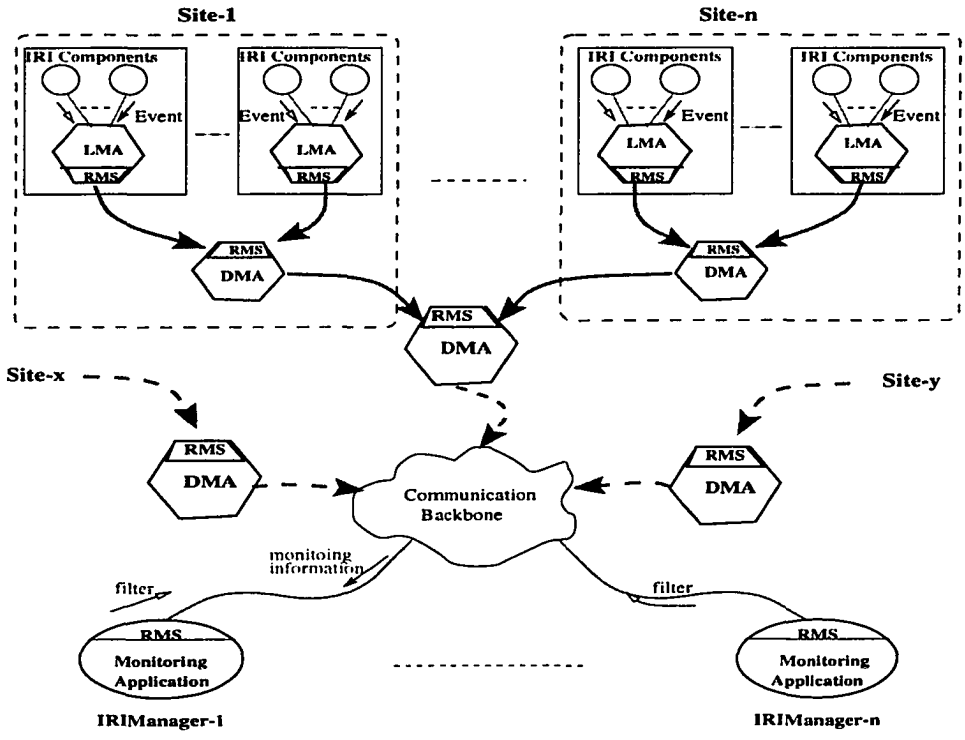


Fig. 7.2. The Monitoring Architecture in IRI Sessions.

ment. IRI components send events to the LMA via UNIX Sockets [86]. A DMA is also assigned for each site in the environment. MAs may also exchange domain information using RMS in order to detect any composite events in the session (i.e. all sites). Therefore, a two-level monitoring hierarchy is found to be sufficient in the current IRI environment. As described in Section 4.2, IRI event consumers send filters to the LMA which parses, decomposes and allocates filtering tasks to other MA(s). In addition, IRI event producers send event notifications to LMAs that perform filtering tasks and forward detected primitive events to DMAs.

7.2 IRI Monitoring Applications

The following describes "real" examples of monitoring applications used in IRI. This includes three major applications: *debugging and testing*, *applications steering*, and *fault recovery*. More applications and examples can be developed using the same technique and guidelines outlined in this section and in Appendix A.

7.2.1 Debugging and Testing

Debugging distributed applications is a complex task because events, such as errors that occur during execution, are concurrent and occur throughout the system. This section presents how the monitoring system can be used, step-by-step, to effectively test and debug Send/Receive functions in the RMS component.

The sending and receiving of messages using RMS is a major activity in IRI. In such message-passing applications, message size mismatches are likely to occur either as a result of operational errors or due to incorrect word alignment of sent packets[†]. Therefore, testing "send" and "receive" operations is highly desirable in any message-passing distributed application such as IRI.

The testing scenario outlined here is referring to a "message size mismatch" scenario. In this scenario, monitor send and receive events in RMS and report information about identified size mismatch conditions. In this monitoring example, the consumers are the developers while event producers are RMS entities. A composite event situation is encountered since knowledge of sent and received message sizes is distributed throughout the IRI environment. The following describes constructing and processing of monitoring activity within the IRI environment.

• **Event Specifications in HESL:** The send multicast event called MSend and the received multicast event called MRec may be specified respectively as follows (see Table 3.2 for HESL syntax):

EVENT= { `ModuleName`=RMS, `FuncName`=Send, `Immediate`;
`IPdest`=224.*.*.*, `IPsrc`=ANY, `seq`=ANY, `size`=ANY } `MSend`.

[†]some kernels or compilers insert extra bytes to make the packet length in multiple of words.

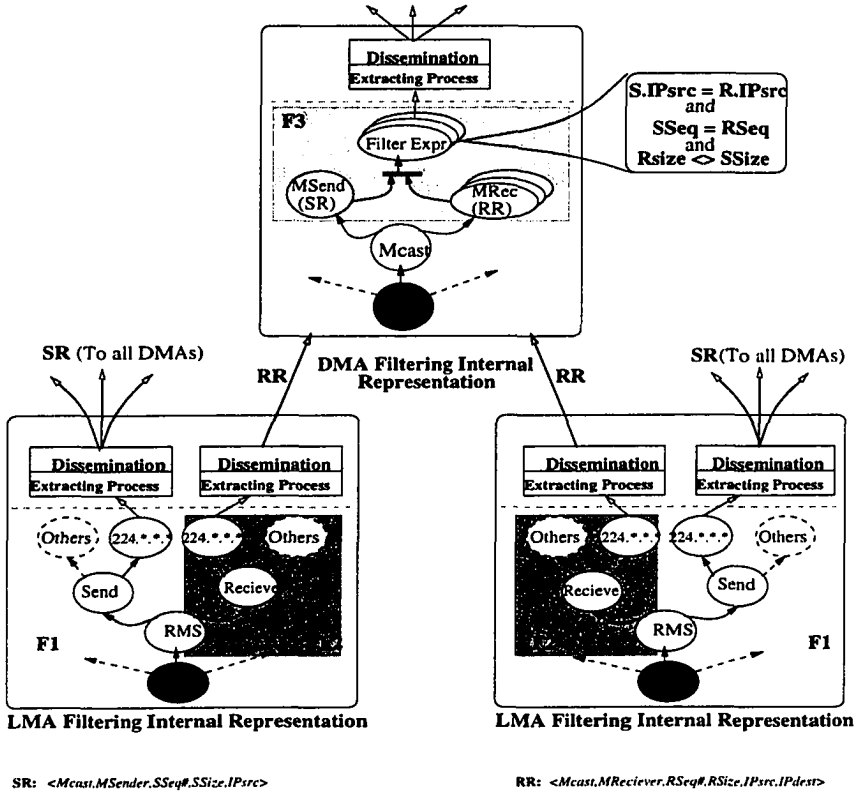


Fig. 7.3. RMS Debugging Example.

EVENT= { `ModuleName`=RMS, `FuncName`=Receive, `Immediate`;
`IPdest`=224.*.*.*, `IPsrc`=ANY, `seq`=ANY, `size`=ANY } `MRec`.

• **Filter Subscription in HFSL:** A developer may define his/her filter subscription as follows (see Table 3.1 for HFSL syntax):

FILTER= $[(MSend \wedge MRec)]$;
$[(MSend.IPsrc{=}MRec.IPsrc \wedge (MSend.seq{=}MRec.seq \wedge (MRec.IPdest{=}124.*.*.* \wedge$
$(MSend.IPdest{=}124.*.*.* \wedge MSend.size \neq MRec.size))))]$;
$[FORWARD]$; `Msg_Mismatch_FILTER`.

`MSend` and `MRec` are the multicast send and receive events. The action is to `FORWARD`, which means that monitoring information is sent to the developer reporting the occurrence of this event.

• **RMS Instrumentation:** The RMS send and receive routines are instrumented to report information about any send and receive events according to the following event specifications: event name, message sequence number, IP source address and size of sent and received message. These fields are used in the filter definition (program).

• **Monitoring Process:** Figure 7.3 shows the internal filtering representation after the filter has been decomposed and distributed between monitoring agents. The filter is decomposed into three subfilters: *F1* detects receiving multicast events and forwards them to its DMA, *F2* detects sending multicast events and forwards them to *all* DMAs, *F3* evaluates the filter expression by comparing the receiving and the sending multicast events.

If the composite event represented by `Msg_Mismatch_FILTER` is detected, then *F3* will forward the monitoring information to the developer(s) as requested. Note that sending and receiving events can take a place in any machine (RMS resides on every machine in the IRI application). Based on the Environment Specifications, the F1 and F2 subfilters are delegated to all LMAs in the IRI environment. However, F3 is delegated to the DMAs, which get `MSend` and `MRec` events and evaluate the filter expression accordingly.

The extracting layer shown in Figure 7.3 forwards relevant information (SR, RR) and reduces event traffic. Finally, the requested monitoring information is forwarded to one or more developers based on their subscription. This simple example shows how developers can effectively monitor and test IRI functions, in general, by collecting and correlating events from various locations in the application environment at run-time. This enables the developer to specify testing and debugging demands without analyzing multiple file traces or inspecting the application entities physically at different locations. Using the same example, developers can limit their monitoring/testing demands on a specific multicast group by specifying in the existing filter $MSend.IPdest = 124.x.y.z$ such that $124.x.y.z$ is the multicast address of this group.

7.2.2 Customizable Event Traces

Generating and collecting event traces in distributed systems is a very useful technique for studying and analyzing the run-time behavior of such systems. For this reason, the distributed systems research community has given considerable attention to this subject [11, 13, 33, 35, 38, 50, 54]. This process is also referred to, in literature, as collecting the "event history". Constructing an event trace or history facilitates debugging distributed programs through use of the following techniques:

- *Browsing event history*- Event traces can be examined through the use of specialized tools ranges from text editors to visualization tools. This examination process enables checking program states and variables at various execution stages.
- *Reply/Visualize program execution*- Some debuggers use event history to control a re-execution of distributed programs. This enables developers to use traditional debugging techniques such as break points and state stepping without affecting program behavior.
- *Simulate program execution*- A collection of event traces can be used to simulate the program run-time environment for any single process. This enables using sequential debuggers without re-executing the entire application.
- *Multiple views*- Event history allows different programmers to have multiple views of a running distributed application. Each programmer can extract the event history/traces related to his/her development task.
- *Future analysis*- Event history also benefits studying the performance and tuning of distributed programs through analysis of demonstrated behaviors.

Three techniques are proposed to generate event traces:

1. *Passive snooping*- In this technique, the event history collector is a program that promiscuously monitors the activities of the communication bus. This technique has limited application since it is not feasible to monitor the communication bus in many of today's distributed systems. Furthermore, monitoring the communication bus may not sufficient to convey all activities performed in distributed applications.

2. *Inserting recording instructions in the program code*– This technique requires instrumenting the program with instructions that record all activities and program behavior. Various literature argued the case that this process may cause considerable impact on program execution. Therefore, it is not recommended for most critical distributed applications.

3. *Instrumented system calls*– In this technique, the operating system performs calls, or supported libraries are instrumented, so that original services are conducted while recording program execution [50]. Despite of the minimal intrusiveness of this technique, some distributed environments prohibit users from changing the OS and its shared libraries. Therefore, in these environments, this technique is not feasible.

The event traces supported by our monitoring architecture provide improvements over the previously stated techniques. A more *efficient* and more *flexible* mechanism for constructing event history in distributed environment is provided.

Although programs are instrumented similar to the second approach, event traces are generated based on user specifications. This means that, unlike the second technique described above, event traces can be customized based on event name, type, source and other information. Dynamic signaling, and hierarchical filtering permit recording of specified events only thereby substantially minimizing intrusiveness. Customizable event traces offer tremendous flexibility in debugging large-scale distributed systems through support of the following features:

- *Dynamic and centralized control of event traces:* Consumers can dynamically define trace specifications and request that data be forwarded to a centralized machine or logger service. Centralized control is highly valuable in distributed environments especially in large-scale systems. Figure 7.4 shows examples of various trace filters. The `TraceAll` filter enables collection of all application generated events and forwarding them to the requesting consumers.

- *Customizable event traces:* Consumers can elect to trace events by module names, location, or event type. This enables consumers to customize the type of event history/trace and to customize information content. For example, in Figure 7.4, `TraceWarning` filter enables tracing all warning events generated from any process

EVENT= { ModuleName=ANYTHING, FuncName=ANYTHING, Immediate;
Machine="ANY", Type="ANY", Info="ANY" } `AnyEvent`.
EVENT= { ModuleName=XTV, FuncName=ANY, Immediate;
ToolName="ANY", Status=Started } ToolStarts.
EVENT= { ModuleName=XTV, FuncName=ANY, Immediate;
ToolName="ANY", Status=Terminated } ToolStops.
EVENT= { ModuleName=ANYTHING, FuncName=ANYTHING, Immediate;
Machine="ANY", Type="Error", Info="ANY" } ErrorEvent.
FILTER= [*AnyEvent*];
[*TRUE*]; [FORWARD]; `TraceAll`.
FILTER= [*AnyEvent*];
[(*AnyEvent.ModuleName* = " * " ∧ *AnyEvent.Type* = "*WARNING*")];
[FORWARD]; `TraceWarning`.
FILTER= [*AnyEvent*];
[(*AnyEvent.ModuleName* = "*XTV*" ∧
(*AnyEvent.Type* = "*WARNING*" ∧ *AnyEvent.Machine* = "*dragon*"))];
[FORWARD]; `TraceXTVdragonWarning`.
FILTER= [*ToolStrats*];
[*ToolStarts.ToolName* = "*Netscape*"];
[*ADD TraceAll*]; `StratXTVTrace`.
FILTER= [(*ToolStops* ∨ *ToolStarts*)];
[(*ToolStops.ToolName* = "*Netscape*" ∨ *ToolStarts.ToolName* = "*Emacs*")];
[*DEL TraceAll*]; `StopXTVTrace`.
FILTER= [*ErrorEvent*];
[(*ErrorEvent.ModuleName* = " * " ∧ *ErrorEvent.EventType* = "*Error*")];
[*ThisMod* = *ModuleName*;
MOD TraceALL.FX = [*ANYEvent.ModuleName* = *ThisMod*]]; `DynamicErrorTrace`.

Fig. 7.4. Customizable and Dynamic Event Traces Examples.

in the IRI environment. However, the `TraceXTVdragonWarning` filter enables tracing only *Warning* events that are generated by the `XTV` [1] module in *dragon*. Similarly, consumers can limit the scope of tracing dynamically, and with minimal overhead in the application environment.

- *Controlling event trace timing:* Consumers can specify start and end times for any given trace activity. In other words, consumers can specify to start/stop a trace activity when a certain event (primitive or composite) is detected. This is a useful in minimizing trace effect and for producing concise traces. For example, the `StratXTVTrace` filter uses filter incarnation (see Section 3.1) to activate (start) the `TraceAll` tracing filter only when the "Netscape" tool is started. Similarly, the `StopXTVTrace` filter permits terminating `TraceAll` when either "Netscape" terminates or "Emacs" starts.

- *Dynamic Tracing:* Traditionally, trace specifications are static and defined prior to any monitoring operation. However, in dynamic traces, the trace specification can be determined during the monitoring process itself based on event patterns and information. For example, developers may want to generate an event trace/history for processes that have produced at least one error event, `ErrorEvent`. In this case, the module name is not known to the monitoring system at trace specification time and the monitoring system must determine the module name. This can be achieved via dynamic tracing supported in the monitoring architecture. Dynamic tracing, one of the novel features of this monitoring architecture, utilizes filter incarnation and filter registers in order to track and restore event information. In Figure 7.4, `DynamicErrorTrace` shows the filter specification for dynamic trace mentioned above. Notice that *ThisMod* is a filter register that restores the module name, *ModuleName*, after the occurrence of `ErrorEvent`. Then, filer incarnation is used to modify the expression of an active filter, *TraceAll*, so that the modified filter executes the new trace specifications.

The hierarchical filtering architecture enables event traces to be combined, reduced or processed during the monitoring process. These can be specified as "actions" performed by LMAs and DMAs during trace collection. The compression and processing activities can result in smaller history sizes with better event presentation.

Utilizing the distributed agents hierarchy, event traces can be processed in a distributed and concurrent manner, which results in a high-performance and scalable distributed event tracing mechanism. This is dramatically different from centralized event trace mechanisms proposed by previous work such as [93]. Although the focus of this research dissertation is not to outline elaborate solutions for processing event traces, this discussion is sufficient to present the potential of building advanced, efficient event traces and logger services using this monitoring architecture.

7.2.3 On-line Application Steering: Slow Clients in Reliable Multicasting

Reliable multicasting is a fundamental component in interactive distributed multi-party systems such as IRI [56]. IRI uses Reliable Multicast Server (RMS) described in [4] to deliver reliable group communication to its tools. One of the major goals of using a HiFi monitoring system in IRI is to collect statistics on the RMS performance, which enables steering and tuning of RMS at run-time. The integration of RMS and management capabilities significantly improves RMS performance, thereby improving the QoS attained by large-scale distributed applications such as multi-party communications.

One of the problems that experienced with RMS that uses token-based Reliable Multicast Protocol RMP [92], is the effect of slow members (e.g., machines) in group communication. A machine is described as a slow machine if its receiving rate is "much" less than that of other members in the group. This, for example, could be the case if a machine is overloaded with jobs or has low hardware capabilities. In this case, a slow machine typically slows down communication in the entire group. In RMS, the sender transmission rate eventually adapts to the rate of the slowest receiver, even though other members are capable of handling higher transmission rates. This causes a serious problem in interactive distance learning applications, such as IRI, where machines in various sites could have different configurations and capabilities [56]. Some of the solutions proposed to handle slow members include: (1) Disconnecting slow members from IRI system completely, or (2) Isolating the slow members by transferring them to a different multicast group that obtains a lower quality of service (e.g. low video quality) in the same session which reduces the load and improves the receiving in these machines [71].

Developing a solution for slow members in multicast groups is beyond the focus of this research dissertation. However, the effective use of HiFi is presented in context of *dynamic discovery* of slow members (machines) during an IRI session and the *automatic feedback* to RMS senders, which accordingly make the proper management decision. The criteria for slow members is defined based on user specifications. For example, the user (or manager) may define a slow member whose performance is below a certain threshold relative to other members. In the example below, the RMS sender acts as a manager and sends the threshold information. Figure 7.5 shows the event (HESL) and the filter (HFSL) specification used to discover slow members in IRI. Each RMS receiver is instrumented to send the *McastRec* event that contains the machine name, the domain name, total bytes received (KBrec), and number of NACKs scheduled (NackSch)[‡]. The `McastRec` event could be sent periodically based on time limit or maximum number of after maximum number of bytes is received. In IRI, RMS receivers send `McastRec` event after each 512K bytes received. RMS senders send `McastSend` to indicate the drop rate (threshold) in the group. However, because the threshold value is dynamic and may be determined from the overall performance of the participants, another filter (see Figure 7.5) is used to provide feedback on the overall drop rate average to senders. Consequently, the threshold value is readjusted accordingly.

Each LMA forwards `McastSend` and `McastRec` primitive events to its DMA, which evaluates the filter expression upon receiving both events. Figure 7.6 presents the filters in Petri Nets (PN) representation as constructed by DMAs. The `Slow_Members_FILTER` waits to receive one `McastSend` and `McastRec` event from all LMAs in the domain. Then, the filter expression is evaluated. The `_ctr` and `_LMAs` are HiFi reserved key words used to denote the number of event occurrences and the number of LMAs in the domain, respectively. The number of event occurrences is represented by `_LMAs` and `_DMAs` in the PN arch. The filter expression results in *true* if all RMS receivers in the domain send `McastRec` events and the NackSch of one or more receivers is higher than the threshold. If the filter expressions becomes true, then three actions are performed: (1) the average scheduled Nacks for receivers in same domain is calculated (CalcAVG), (2) the `DomAVG`

[‡]*NackSch = Number of NacksSent + Number of NacksCancelled.* Because of the NACK suppression mechanism [24], the number of *NackSch* gives more accurate estimation of the *drop rate* than number of Nacks sent.

EVENT= { ModuleName=RMS,FuncName=McastSend,Immediate;
Machine="ANY",Domain="ANY", threshold= ANY } McastSend.
EVENT= { ModuleName=RMS,FuncName=McastRecv,Immediate;
Machine="ANY",Domain="ANY", KBrec= ANY, NackSch=ANY } McastRec.
EVENT= { ModuleName=DMA,FuncName=ANY,Immediate;
Machine="ANY",Domain="ANY", KBrec= ANY, NackSch=ANY } DomAVG.

FILTER= $[(McastSend \wedge McastRec)]$;
$[(McastRec.\_ctr = \_LMAs \wedge McastRec.NackSch > McastSend.threshold)]$;
$[CalcAvg, DomAVG$, FORWARD]; Slow_Memebrs_FILTER.

FILTER= $[DomAVG]$;
$[DomAVG.\_ctr = \_DMAs]$;
$[UpdateThrehold, McastSend]$; Update_Threhold_FILTER.

Fig. 7.5. HiFi Application Steering Example.

event is sent to the containing DMA to reveal the domain average, (3) the McastRec event that matches the slow member criteria represented in the filter expression (i.e., NackSch > threshold) is forwarded to the manager (RMS sender).

The second filter, Update_Threshold_FILTER, receives the DomAVG events from the DMAs and then calculates the total NackSch average, updates the threshold and sends the McastSend containing the new threshold to the LMAs/DMAs again. This filter can be a DMA task, instead of RMS senders. However, users must provide the action *UpdateThreshold* to this DMA. The DMA will then dynamically update the threshold while RMS senders manage the slow member problem.

Slow members and NackSch average information is collected from each receiver via LMAs and then combined and propagated in hierarchical fashion, via DMAs, to the RMS of the sender. In addition to offering the dynamic information feedback service, this mechanism is scalable as it avoids notification implosion that typically occurs when McastRec events are forwarded to one RMS sender. Furthermore, distribution of processing load

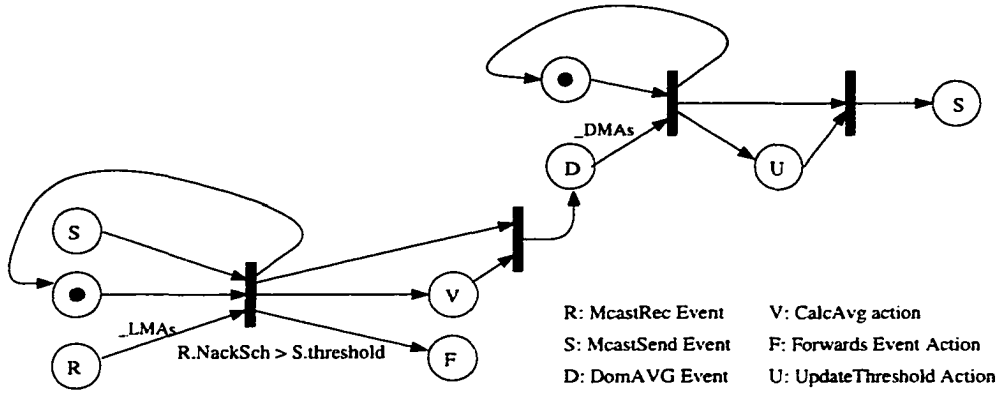


Fig. 7.6. Steering Filter in PN Representation.

(e.g., the calculation of the average drop rate) contributes to the system performance.

7.2.4 Fault Recovery

Providing a fault detection and recovery mechanism is essential to improving reliability and robustness of an large-scale distributed system such as IRI. In [9], many types and sources of failures that can occur during IRI execution and which reduce quality of service in the IRI session, were discussed. In this section, fault recovery mechanisms provided by the monitoring architecture to improve reliability and robust of the IRI system is outlined briefly.

Monitoring Functions for Fault Recovery: Effectively observing error and warning messages revealed by IRI components during execution is not feasible. This is due to the large number of messages that are generated at run time and the geographical distribution of the component entities. Monitoring is essential to classify and detect application errors or failures as they occur Manual or human recovery may be insufficient to resolve problems because of the natural delay incurred in the human interaction [9]. In IRI, the monitoring system is used to detect and automatically recover from failures, as represented by error and warning events, without involvement of IRIManagers or developers. Moreover, IRI developers can specify different priorities for errors and warnings, so that failure events are processed based on their priorities. This is an important criteria for managing multimedia

services. For example, in IRI, the audio service is more important than other services, such as video service. For this reason, audio events are assigned a higher priority than other services, which provides a more immediate response to audio failures.

Errors and warnings that may occur during program execution are sent as event notifications to the MA(s). When an event is detected, a corresponding pre-defined action will be performed. In an IRI monitoring environment, an action is either (1) a program executed by the MA or, (2) a notification sent by an MA to the requesting IRIManager, who consequently takes the proper action to recover this fault. In the following, we discuss the two classes of fault recovery services supported in IRI.

• **Automatic Fault Recovery:** The *automated fault recovery* mechanism is supported to initiate the proper recovery procedure upon fault detection, which limits impact to users and administrators (IRIManagers). To use automatic fault recovery, IRIManagers or developers first must develop the action program that is executed when a specific error event is detected. Secondly, IRIManagers must attach this action to specific events in the HESL by specifying the action program name. Since IRI failure events (errors and warnings) are already specified, IRI developers or IRIManagers need to determine and develop the recovery actions and attach them to the failure events. When the monitoring system detects faults events in IRI, it initiates the corresponding recovery procedures (e.g., performing `fork` and `exec` on the action programs).

The automatic recovery may independently occur in any local machine. When this occurs, the recovery is referred to as *local fault recovery*. An example of local fault recovery in IRI is the automatic restart of crashed components (audio, video, presentation tool), which occurs independently of other machines and entities in the classroom. An LMA detects the crash of a component through receipt of a UNIX `SigPipe` signal, or from notification from the SC. At this point, the LMA will restart the IRI component by executing the appropriate local recovery procedure [9]. As a result, notification of the failure and recovery is sent to the teacher.

However, since some faults impact the entire session, the recovery procedure may need to be initiated and coordinated throughout the classroom. This type of recovery is referred to as *global fault recovery*. Failure of IRI components, such as TSE, require global fault recovery procedures. Restarting a TSE entity independently of the other TSE entities in the IRI environment causes inconsistency in the system.

EVENT= { ModuleName=PRESENT, FuncName=SendSlide, Immediate;
Machine="ANY", SlideNum=ANY } SendSlideEvent
EVENT= { ModuleName=PRESENT, FuncName=RecSlide, Immediate;
Machine="ANY", SlideNum=ANY } RecSlideEvent.

FILTER= $[(SendSlideEvent \wedge RecSlideEvent)]$;
$[(SendSlideEvent.SlideNum = RecSlideEvent.SlideNum \wedge RecSlideEvent.\_ctr = ALL)]$;
$[FORWARD]$; `SlideSynch`.

Fig. 7.7. Event Correlation for Multimedia View Synchronization.

• **Manual Fault Recovery:** The manual fault recovery mode is important for failures that require human intelligence and experience. Manual recovery is used when the *action* is forwarding (`FORWARD` in HFSL) the monitoring information to IRIManagers. The monitoring system enables IRIManagers (or developers) to centrally collect and observe IRI run-time failures (errors and warnings events) that occur at different places in the IRI environment from one location. More effectively, IRIManagers have the capability to focus and control the granularity of monitoring demands based on many parameters, such as machine name, component name, module name, event type and event priority. For example, the IRIManager may desire to monitor *only* `Major_Error` and `Minor_Error` IRI events. Or the IRIManager may request to monitor *only* the failure events of the Audio component of the teacher machine.

Collecting, suppressing and forwarding failure information, such as error events, to a central point is significantly useful for generating global software traces and for manually recovering from these failures (e.g. `rsh` to remote machines and fix the problem).

From our experience in IRI, both approaches (automatic and manual) were found to be important to recovering failures in the IRI sessions.

7.2.5 Event Correlation for Multimedia View Synchronization

In interactive distributed multimedia applications such as IRI, different multimedia streams may interleave independently because of jitter and network delays [22]. This may cause same events (views) to be received and displayed at different times by receivers in such

distributed environment. IRI is a collaborative application that enables global sharing of some applications. For example, the teacher or presenter may use the *Slide Tool (ST)* application to present a slide show to students in the classroom session. A presenter may also use the *global pointer*, or chalk board, to emphasize an issue in a slide. In addition, presenters use audio to explain and elaborate on ideas and issues in the displayed slides. Three different, but temporarily related, activities (slide display, pointer movements, audio) are involved to deliver a slide show presentation in a typical distributed multimedia system environment.

To improve the classroom interaction, it is important for a presenter to synchronize delivery of various information streams (data, pointer or audio) with the view of the students (or audience). Synchronization of multimedia streams is not in the scope of this work, however, providing an opportunity to demonstrate effective event correlation and feedback to solve this complex problem.

Figure 7.7 illustrates the `SlideSynch` filter that detects a correlation between the events of displaying the slide in the presenter machine, *SendSlideEvent*, and that in the remote machines, *RecSlideEvent*. When `ALL` receivers view the same slide number, *SlideNum* , the filter sends notification to the presenter. The presenter is the consumer in this case and can request notification as to when all or subset of receivers are viewing the same slide by assigning the proper value to `_ctr` in the filter expression. This feedback information enables the presenter to synchronize information delivery (e.g., start talking or moving the pointer on the slide when all receivers are simultaneously viewing it).

7.3 Summary

This chapter describes some applications of HiFi monitoring system. Examples are presented as applied in IRI distributed distance learning system. These applications include debugging and testing, generating distributed traces, applications steering, fault recovery, and view synchronization in distributed multimedia systems.

In debugging and testing, we show how HiFi can be employed to detect certain application errors/bugs. The presented example was to detect the message size mismatch problem between the sender and receivers in distributed systems. We then show how HiFi can be effectively used to collect and customize traces in large-scale distributed

environment. Using HiFi, users are able to change the trace specifications at run-time in order to limit the monitoring scope and focus their observation. The filter incarnation (see Section 3.3) enables users to define "dynamic traces" filters that can dynamically modify the trace specifications based on the application events generated at run-time. Various trace filters examples are presented in this chapter. We then describe an example of HiFi application steering for distributed multi-point applications or reliable multicasting. In this example, HiFi identifies slow members in the multicast group based on the average Nacks and a user defined threshold. Two filters are used in this example. The filter one is used for comparing the members' Nacks against a threshold, and the other one is calculates the Nacks average per domain. Both filters are integrated to discover the slow members in a multicast group. HiFi is also used to support manual and automatic fault recovery in large-scale distributed systems. Users can use HESL and HFSL to specify event correlations (pattern) identified as system faults, and associate actions to be performed when such correlations are detected. The last presented application describes how HiFi is utilized to discover if all participants in a distributed multimedia session are simultaneously viewing the same data (e.g., slide or pointer position). This feedback in important for synchronizing the presentation activities in distributed multimedia applications.

CHAPTER VIII

RELATED WORK

Although a large number of monitoring systems were identified, this thesis concentrates on monitoring systems that address requirements and challenges of monitoring LSD systems, as described in Chapter 1. As our approach emphasizes the impact of the event filtering mechanism as a major component in the monitoring architecture, our related work includes both *monitoring distributed systems* and *event filtering*. The following survey study is divided into two parts. The first part (Section 8.1) addresses monitoring related work while the second part (Section 8.2) discusses filtering mechanisms. In this Chapter, monitoring types are introduced and defined as they pertain to distributed systems. Examples are presented and evaluated in terms of monitoring LSD applications. We also compare these systems/approaches with the HiFi monitoring approach.

8.1 Survey and Evaluation of Monitoring Distributed Systems

Existing monitoring systems are classified according to three major approaches, *hardware monitoring*, *software monitoring* and *hybrid monitoring* with a comparison made of each. This section briefly describes these systems and compares and evaluates them against HiFi. Monitoring distributed systems is classified into three main categories:

8.1.1 Hardware Monitoring

In [36] and [57], several hardware monitoring systems were presented. In this class of monitoring, specialized hardware is dedicated to observe the monitored system and detect interesting events. The hardware performs event detection by snooping into the system

bus or the network media (such as *Sniffer* [34]), or by connecting physical probes to the processor, memory, ports and/or I/O channels. Hardware monitoring systems have the advantage of being non-intrusive, and being efficient and accurate. Typically, this is accomplished via a special purpose device that consists of an external, independent system environment (e.g., its own processor and memory). This external hardware limits the amount of interference with the monitoring application environment as it is dedicated to monitoring functions. Hardware monitoring is particularly important to enable effective real-time monitoring and is critical when monitoring of hard real-time is required. However, hardware monitoring has the following disadvantages:

(1) Hardware monitoring systems, such as Sniffer, are usually restricted to monitoring the connected system or object (e.g., network subnet). This limits the scope of the monitoring process and makes monitoring LSD or distributed applications difficult.

(2) Hardware monitoring is more expensive since special hardware components are required.

(3) Control is difficult without a complete software environment (e.g., OS). Adding a complete software environment adds additional expense and decreases system flexibility. (4) Portability is limited and often expert personnel are required to install and maintain the monitoring environment. This again increases costs and decreases flexibility.

8.1.2 Software Monitoring

This class of monitoring systems uses software programs running in the same environment as the monitored objects. This requires that system resources (e.g., processor and memory) be shared. Monitored programs are usually instrumented by inserting monitoring instructions called *software probes* to gather information. The main advantages of software monitoring systems are: (1) flexibility of use since they are provide easier construction and control processes, (2) portability to other system or platforms, (3) maintainability since the basis of the monitoring system is program code, and (4) economy since no special hardware devices are required. However, software monitoring may suffer from the impact of sharing system resources with the monitored objects. This may decrease the accuracy and performance of monitoring while increasing intrusiveness. For this reason, pure software monitoring systems are insufficient for hard real-time monitoring.

Software monitoring systems are discussed below with specific examples given. Major

limitations of each, from the perspective of monitoring LSD systems, are outlined.

Standard Monitoring Tools: The two existing management standards are SNMP (Simple Network Management Protocol) [18] and CMIP (Common Management Information Protocol) [82] ISO/IEC 9596-1. While SNMP is currently the de facto standard for managing the Internet, CMIP has been hailed as the long-term successor for network management protocols. SNMP is a simple management protocol that uses polling requests (`Get` and `Set`) and *traps* to represent extraordinary events for monitoring and management operations. SNMP has limited scalability potential that does not permit SNMP to effectively monitor distributed systems [72]. SNMP also utilizes UDP, an unreliable communication service, for message delivery. In contrast, CMIP is a much more complex protocol and uses event-driven techniques in the management operation. As discussed in Section III, the event-driven approach is more efficient in monitoring LSD systems. The complexity of CMIP implementation may significantly impact its performance and could increase the intrusiveness of the monitoring process (the OSI Event Report Management is discussed in Section 8.2). SNMP and CMIP are primarily intended for monitoring system and network objects (such as machines, routers, bridges) rather than monitoring applications or distributed applications. Every monitored attribute has to be maintained in the Management Information Base (MIB) which must be synchronized and consistent in a distributed environment.

Issos System: This monitoring system was developed as part of the Issos parallel programming system at the Ohio State University. The Issos system provides dynamic real-time and application-dependent monitoring for parallel (i.e., multiprocessor) and distributed (i.e., cluster of workstation in LAN) systems. Users can specify time constraints for monitoring operations and change the values of the monitoring attributes at run-time. This is useful for on-line debugging and application steering [69]. Monitoring specifications are defined via the entity-relation (ER) model [20]. One major advantage of this system attribute of this monitoring system is its flexibility in providing a wide-range of collection mechanisms, such as probing sensors, tracing sensors and extended sensors (with analysis in the program sensors). The Issos monitoring system has a number of limitations when monitoring large-scale distributed systems.

- Issos monitoring uses a semi-distributed monitoring approach with monitoring operations distributed among a *centralized monitor* and *resident monitors*. The centralized monitor is located in a remote node of the LAN and receives notifications from resident monitors for correlation and final evaluation. Resident monitors reside in each node of the same connected LAN. This filtering architecture does not scale well with respect to application entities or manager requests.

- Discussion about environment specifications or how the monitoring agents (residents monitors) are distributed in the system are not provided. Therefore, it is presumed that these activities are manual in nature and are the responsibility of the user.

- Although this monitoring system provides hooks for "action" specification, identification of the rich set of actions needed for system adaptation was not provided.

- Dynamism is supported by permitting monitoring variables to be changed at run-time. However, this support is insufficient when the monitoring expression itself must be modified or another one must be added. Adaptation/reconfiguration must be performed explicitly by the steering program. Adaptation can not be part of the monitoring specification itself, nor can automatic activation occur as is the case with HiFi filtering incarnation.

- The Issos authors state that monitoring queries must be statically specified, optimized, and compiled into the application itself prior to application execution [69]. This is a significant limitation of monitoring dynamism when compared to HiFi, which permits filter and event specifications to be added, deleted and modified dynamically at run-time without the intervention of the monitored system itself.

- In this monitoring implementation, LAN nodes and machines running distributed applications are assumed to have clocks synchronized with microsecond accuracy.

- The monitoring system supports limited monitoring/filtering optimizations, such as housing the expression evaluation in close proximity with the application to reduce communication overhead. However, support is not provided for placing evaluation in domain agents between the applications or resident monitors and the central monitor. In distributed systems monitoring, the domain agent is often the optimal place for

evaluation to occur. HiFi permits this optimal placement of the filter expression to be determined during the filter decomposition and assigns the proper DMA which reduces communication overhead and maintains monitoring scalability. In addition, HiFi supports other optimization techniques not supported by the Issos monitoring system. These include efficient filter composition and matching techniques.

Falcon System: Falcon is an on-line monitoring system developed at Georgia Tech for steering large-scale parallel programs [32]. It supports sampled sensors, traced sensors, and traced sensors with filtering and analysis capabilities [78]. Falcon has three major attributes: dynamic overhead control by providing configurable system parameters (such as buffer length), application-specific monitoring, analysis and display features, and scalability large-scale parallel programs running on large numbers of parallel processors. One important feature of Falcon is its ability to be reconfigured at run-time to meet the needs of the application [32]. The Falcon system faces the following limitations with regards to monitoring large-scale distributed systems:

- The Falcon architecture includes decentralized filtering, shared memory communication, and dynamic thread forking. Thus, making it more appropriate for parallel programs in a multiprocessor configuration rather than that of a distributed system running on interconnected workstations.
- Although Falcon is used for application steering and adaptation, it seems that event correlation is not supported. Adaptation may require feedback information from two or more threads (nodes) in the application environment.
- Filtering performed by extended sensors is very simple (not expressive) and, since events are processed in the same environment, considerable overhead is inflicted on the running application.
- Unlike HiFi, Falcon does not provide a complete monitoring environment such as automated instrumentation or environment and action specifications.
- Falcon presents considerably high perturbation when full event trace is requested. However, Chapter 6 shows that HiFi incurs less and more controlled application perturbation than Falcon.

Meta System: The Meta monitoring system is a collection of tools used for constructing distributed application management software in conjunction with the ISIS distributed toolkit citeBirman:94a. Meta enables management applications to observe and control functional behavior of monitored programs. To manage distributed applications using the Meta system, three steps must be performed.

(1) *Instrumentation:* The monitored program is instrumented by inserting *sensors* and *actuators*. A sensor is a function that returns program state and environment values (i.e. cpu load) while an actuator is a function that changes variables or larger portions of a running program to control its behavior. Sensors and actuators can operate on the application or its environment. For example, a built-in actuator can change processing priority. Both can be specified using a rule-based control language called *Lomita*. The Meta system uses either on-demand or periodic polling requests to obtain information about the state of the monitored program.

(2)*Program Structure Description:* The programmer/developer must describe the structure of the monitored program using entity-relationship database concepts. For example, each component/function of the system can be represented as an entity. This model of the program abstraction is used by the *Policy Layer* to interpret and perform management functions within the control component of the Meta system.

(3)*Policy Rules:* Using Lomita, the programmer then uses the data model to write a set of policy rules that specify the desired system behavior. The programmers may make direct calls to sensors, actuators or other functions defined in the data mode [58]. The Meta monitoring system has the following limitations:

- Sensors are static programs that are linked with the monitored application prior to its execution. This reduces the dynamism and the flexibility of the monitoring system.
- The program structure must be declared to Meta using an object-oriented model prior to any monitoring request. This requirement could be inconvenient for other monitoring applications, such as debugging and testing, because the program structure frequently changes as bugs or errors are discovered and corrected.
- Non-local sensors are accessed remotely by the MetaLib attached to the application

itself. This means that event correlation is always performed in the MetaLib assigned for executing Lomita control script. Perturbation is increased and monitoring performance and scalability are reduced due to centralized event correlation.

- The Meta system uses a decentralized filtering architecture since sensors are only located in MetaLib(s), which are attached to each running application entity.
- Unlike HiFi, the instrumentation process is handy and imposes a considerable overhead on event consumers. Also, environment specifications is not supported which implies an overhead in administrating and distributing the Meta agents.
- Optimization techniques may be limited as no discussion or description was provided.
- Meta uses the Isis system for atomic group communication, which may reduce the Meta usability as a general monitoring system on UNIX platforms.

Hy^+ Systems: Hy^+ [46] was developed at the University of Toronto to support an query-based visualization interface for network management and distributed debugging. The user can use a declarative language to specify on-demand network management functions or debugging patterns The Hy^+ approach is very similar to the active database approach [93] in which monitoring information is stored in a database and manipulated after collection. The main attributes of the system are the expressive power of its declarative language and its visualization techniques. Hy^+ provides techniques to filter abstract events generated from network elements or distributed systems. Two types of filtering are used: *on-line filters* are integrated with running programs to discard irrelevant events, and *display filters* are used to extract specific information. Hy^+ has the following limitations:

- Although Hy^+ uses decentralized on-line filtering, this kind of filtering is simple. Major filtering and event correlation are performed in the database. This centralized approach does not scale with an increasing number of event producers and consumers.
- The filtering and the abstraction techniques are primitive and do not support many significant properties such as reconfigurability and distributed event correlation. Events can be discarded based on very high abstraction criteria such as process level or event type.

- On-line filters combine event reporting and processing which may increase monitoring overhead in the application environment.
- Although Hy^+ focus on the expressive power, it lacks many other important features such as scalability, dynamism and intrusiveness control. Such issues have not been addressed in descriptions of the system.

Jade Monitoring Systems: Jade was developed at the Calgary University and has been used by a number of universities and research organizations for monitoring distributed systems. Specifically, Jade has been used to monitor the IPC mechanism that occurs during the interaction of distributed applications [44]. This work mainly differs from previously presented systems by requiring the application to generate events for *all* communication activities generated in the application. However, most of the above techniques provide means to specify what kind of events are to be reported. Jade provides interactive, animated displays of executing distributed programs; this enables interactive debugging and control of running programs. In a distributed system, events are generated from one or more monitored programs and sent to *channels*, which are local processes that collect and merge events into an *event stream*. The event stream is then forwarded to one or more *console*. A console examines, interprets and presents the monitoring information to the end users. The system is used for deadlock detection and for debugging and controlling non-deterministic execution in distributed systems. The main limitation in this approach is highlighted below.

- Use of event filtering techniques is not indicated in system descriptions. This will result in extensive monitoring overhead in the application environment, especially when all events are reported as is the case with Jade.
- The instrumentation process is not flexible nor dynamic.
- The primitive event detection is performed in the application. This obviously will cause a considerable intrusiveness in the application.
- The composite event detection (event correlation) is performed in the console (i.e, consumer) which can create a console bottleneck in the monitoring process.

TABLE 8.1

HiFi Comparison with Monitoring Debugging Systems

System	Event Type	History	Filtering	On-line	Inst.	Pert.
dbxtool	Stmt	none	none	No	OS	ns
defence	Stmt	none	na	No	obj	ns
DISDEB	ipc,sh	none	limited	Yes	hw	none
EDL	ns	complete	lang	Yes	ns	ns
HARD	Stmt	none	none	Yes	src	limited
IDD	ipc	buffer	event	Yes	obj	ns
TSL	ipc	complete	lang	Yes	src	ns
HiFi	Stmt	complete	lang	Yes	src	controlled

Stmt: Statement, sh: shared memory, obj: object,
lang: language, event: event name, src: source,

- Scalability is very limited because of the channel and console architecture used in this system.

Monitoring Systems for Distributed Debugging: [60] introduced a comprehensive survey and discussion of monitoring systems used for debugging distributed or concurrent programs. Although HiFi does not provide a complete distributed debugging environment the aim of this chapter is to evaluate the event monitoring techniques (collecting, analyzing and reporting) of such systems and compare them with HiFi approach. Although more than thirty monitoring systems were studied in this reference [21, 60], five of them are considered on-line monitoring systems. These are EDL [13], IDD [33], TSL [35], DISDEB [49] and HARD [54]. Of these, only EDL and TSL systems support language-based filtering where users can specify events of interest. The other three systems (DISDEB, HARD and IDD) either do not support any type of filtering or the filtering is very primitive and limited to function level processes. Neither EDL nor TSL include techniques to control or minimize the probing effect (e.g., monitoring intrusiveness). In addition, none of these monitoring systems address other issues, such as scalability and dynamism, in an efficient way. In particular, EDL and TSL event recognizers represent potential bottlenecks [60]. Moreover, Table 8.1, extracted from a number of tables in [60], shows that HiFi is the only

one among these systems that provides a powerful filtering support (i.e, language) with expressive events (e.g., statement event type) and controlled perturbation technique. and

8.1.3 Hybrid Monitoring

The hybrid monitoring systems attempt to combine the advantages of hardware and software monitoring techniques. Hybrid monitoring consists of dedicated hardware devices for receiving and processing monitoring information. Hence, the monitoring system has its own independent resources but also shares some resources with monitored objects. The main advantage of this monitoring class is that (1) it causes less intrusiveness than pure software monitoring, (2) it is more efficient than pure software monitoring since events are processed in hardware, and (3) it is more flexible and less expensive than pure hardware monitoring.

ZM4/SIMPLE: ZM4 is a hybrid monitoring system [36] that allows monitoring programs to be evaluated for performance and program behavior to be observed. ZM4 consists of *dedicated PCB probe units* that collect and buffer events, *monitor agents* that control the probes and store forwarded events to the disk, and a *control and evaluation unit* that masters the agents and provides the user with more sophisticated analysis tools. The system uses *tick channels* connected to all nodes to support global clock synchronization of DPUs by using physical clocks. Although ZM4/SIMPLE seems to be a powerful monitoring tool for parallel and distributed application, it may not be sufficient for monitoring LSD systems for the following reasons:

- Event detection based on the event information or event correlation is performed in a late monitoring stage (filtering in SIMPLE), which may be an inefficient approach for monitoring LSD environments. This will be discussed later when compared with the approach outlined in this thesis.
- Synchronization via global physical clocking does not scale in WAN (Internet) or large-scale interconnected LAN (Intranet) environments.
- This approach lacks scalability since the monitor agents report to one centralized place where postmortem analysis/filtering is conducted.

- Required hardware modifications for remote or geographically distributed nodes make this implementation highly inflexible.
- Using special hardware makes the system less portable and more expensive.

8.2 Survey and Evaluation of Event Filtering Mechanisms

Work on event filtering spans a number of domains, including distributed system toolkits [39], network and system management [62, 83], communication protocols [12, 59, 63, 94], and active databases [27, 29]. This section describes related work on event filtering , compares and contrasts the different techniques used in these mechanisms and evaluates each one in terms of its support for monitoring.

8.2.1 Distributed System Toolkits

Isis [14] supports event filtering as part of its *Reliable Distributed Objects* (RDO). Isis uses filters as a protection facility to validate authenticated messages in applications such as the Isis Distributed News Service application. Filters are used to distinguish (classify) invalid messages such as unauthenticated messages or truncated messages sent by faulty clients. A message arriving at a consumer is examined by passing it through a series of validation filters. Filters are also used in Isis News to enable consumers in a group to receive all events sent by producers.

Three basic limitations may be found in Isis filtering mechanism: (1) consumer filtering is limited to matching on character strings, "keywords", (2) this decentralized filtering architecture imposes some processing overhead on the consumer since the filtering is performed at the consumer end, and (3) this architecture may also reduce the network utilization (waste of bandwidth) since the filtering is performed at the destination node.

8.2.2 Network and System Management

OSI Event Report Management: OSI Event Report Management Function (ERMF) is described in [83]. ERMF represents a *decentralized* filtering architecture too. Remote management agents in the networks may receive registration requests of Event Forwarding Discriminators (EFDs) which are used to describe events. An EFD contains a filter

expression that describes the fields of a matching event such as event type, event value, and event time. The HP OpenView provides an implementation of ISO OSI Event Report Management Function.

Packet Monitoring Program: The *Packet Monitoring Program* (PMP) [16] is a packet monitoring tool that uses event filtering for gathering statistics about packets in the network and analyzing traffic patterns. The packet parsing mechanism in PMP parses the packet according to the Field Parsing Tree (FPT) which is equivalent to the DAG. Any message field that has to be extracted for statistical analysis must be specified as a node in the FPT. For efficiency, the packet header format is hard-coded in the PMP code at compile time. However, PMP can be configured dynamically since the field values and the statistical rules can be defined at run-time. PMP uses an interpretive pseudo-machine (IPM), discussed in Section 2.2, to define the filter expression.

PMP uses some optimization techniques for efficient filtering composition. If there is a redundant filtering object (i.e. two filters have the same test condition), the Boolean result of the invocation of the first filter is saved and reused when the second filter is invoked. This technique allows evaluating the filtering object without unnecessary re-computation. However, this requires adding two extra fields in the filtering components: the *filter result* and the *indirect pointer*[16].

The filtering technique used in PMP suffer from the following limitations: (1) it does not support detection of composite events, , (2) the filter expression supports only basic operators (`AND`, `OR` and `NOT`) which may not accommodate more complex event filter expressions, and (3) PMP has centralized filtering which makes it insufficient to monitor distributed applications.

8.2.3 Communication Protocols

Several studies have reported measurements based upon various types of *packet filters* (also known as *packet classifiers* [12]). In the following section, we present the evolution of packet filtering mechanisms and an overview of each technique.

CSPF and BPF: The CMU/Stanford Packet Filter (CSPF) [63] and the Berkeley Packet Filter (BPF) [59] were two influential first generation packet filter implementations. CSPF

is a stack-based packet filter that uses pop and push operations. The CSPF uses *Boolean expression tree configuration* and *stack-based interpreter* as the internal representation and the programming interface of the filter, respectively. The stack-based interpreter and tree model limit the performance of CSPF. In contrast, BPF achieves better performance. It uses a register-based assembly language (load and store instructions) and a directed control flow graph (DCFG) instead of stack-based language and tree graph, respectively.

For example, in CSPF each logical operation requires five stack operations (three pushes and two pops) to be executed. This makes it perform poorly compared to the register-based interpreter that uses one simple compare operation (*i.e.*, jeq, jgt). In addition, using DCFG instead of a tree graph model is another reason for the performance difference between BPF and CSPF. A tree model often does unnecessary or redundant computations [59]. Moreover, BPF handles additional features that are not supported by CSPF such as dealing with variable header-length and extracting portions of a packet. Neither CSFP nor BPF support an efficient filter composition technique because the time required to filter packets grows linearly with the number of concatenated filters. Therefore, CSFP and BPF do not scale well with the number of active consumers.

MPF: Similar to BPF, The Mach Packet Filter [94] (MPF) uses DCFG and extended register-based assembly language for the internal representation and filter programming interface, respectively. MPF addresses the scalability limitation exists in the previous packet filters by enabling an efficient composition of multiple filters. MPF achieves more efficient composition than CSPF and BPF by combining the common prefixes of filter predicates. Therefore, unlike CSFP and BPF, evaluating a common prefix is performed just once for any incoming packet passing through a composite filter, regardless of the number of composed filters. In contrast, in CSPF and BPF, the predicate evaluations would grow linearly with the number of composed filters. Moreover, MPF uses hash table lookup to efficiently demultiplex packets to different endpoints.

MPF has a major advantage over previous packet filters. It provides an efficient demultiplexing of incoming messages by combining similar filters (common prefix) together in the filter composition technique. However, this optimization technique does not generalize to a composite event detection mechanism.

PathFinder: PathFinder [12] is a packet classifier that combines software and hardware to optimize the filter composition and dispatching of packets. PathFinder presents a more general technique for composing filters with common prefixes. The software portion of PathFinder builds a directed-acyclic graph (DAG). The DAG nodes (called cells) represent the test predicates and the DAG edges represent the control transfer. The DAG is implemented according to a high-level declarative interpreter that specifies the matching patterns in the packet format. The PathFinder interpreter matches fields of incoming packets using information stored in cells of the DAG. PathFinder may be optimized [12] by re-arranging the cells in the DAG as described in section 2.2.

PathFinder has several novel features that makes it perform better than earlier packet filters such as (1) caching of key-pattern to reduce number of predicate comparisons in the DAG, and (2) provide a hardware support for packet filtering which significantly increase the performance of PathFinder over the previous ones. However, the PathFinder has primary limitations:

- The software implementation of the DAG uses an interpreter, rather than a compiler. This precludes a variety of performance optimizations that can be performed in the compilation process.
- The "dual line"[12] mechanism supported by the PathFinder to deal with IP fragmentation is not sufficient as a general means for composite event (global event) detection.
- PathFinder does not support relational operations (such as `LE`, `GT`). This may limit the application of PathFinder in many domains where these kinds of operators are necessary.

8.2.4 Active Databases

Support for *triggers* is an important distinction between active and standard databases. A trigger is an event-condition-action expression where an event can be either a primitive or composite event. Primitive events in active databases are the basic database operations (such as add and delete records) accomplished in the system. In an active database, event filters are used to detect a composite event since primitive events are already identified

via programming language *exceptions* that are sent whenever a primitive event occurs. A number of approaches have been presented for modeling and detecting composite events in active databases [74]. However, in this section, we discuss briefly three approaches: the COMPOSE system[29], the SAMOS system [27] and the rule and database system [93] (more details can be found in [2]).

The COMPOSE System: The COMPOSE system [29] uses regular expressions and special filter expression operators (called event composition operators) to define event filters. COMPOSE has two types of operators, basic and advanced (examples can be found in [2, 29]). Composite events are detected using Deterministic Finite Automata (DFA) described in Section 2.2.

The SAMOS System: The SAMOS (Swiss Active Mechanism-based Object-Oriented Database System)* uses a modified version of Colored Petri Nets, called SAMOS PN (S-PN), to model and detect composite events. The event filter expression is constructed using basic operators called event composition constructors(examples can be found in [2, 29]). More information about these operators and examples can be found in [27]. The S-PN also uses an incremental procedure to detect a composite event. However, in S-PN, if a primitive event is matched, the place is marked with a token indicating the event occurrence. The "step forward" process continues by advancing the token forward until the last element of the appropriate sequence is marked. This implies that a corresponding composite event is detected [27].

In general, PN representation provides better space complexity than the DFA representation [27].

Rule and Database Systems: In this approach, rule languages (such as *Prolog*) and a database languages are used to define event filters [93]. Events are combined (in a filter expression) using `AND`, `OR` and `NOT` operators only. Parameters and time functions are supported by event filter expression. In addition, the *temporal ordering* between events can also be specified in the event filter expression. For example, the filter: (`Crash-at-5` :- `Crash`, 5:00AM = 20) detects the composite event `Crash-at-5`, if `Crash` event occurs

*This is the name of the prototype active database used in this approach.

within 20 minutes (after or before) from 5:00 AM.

This system uses an off-line event detection process. In other words, in order to detect composite events, the filtering operations are performed on a pre-recorded event history of the system, rather than in real-time. In contrast, the previous systems, COMPOSE [29] and SAMOS [27] uses an on-line composite event detection where primitive events are detected (by exceptions) in real-time immediately after they occur without requiring the recording of the event history [29].

CHAPTER IX

CONCLUSIONS AND FUTURE WORK

In this thesis, the design feature of a monitoring system suitable to large-scale distributed systems, such as those that support Internet-based services and applications, were explored. This work was motivated by the lack of a comprehensive monitoring system that satisfies key requirements in this area.

As inferred from literature on related work, other proposed monitoring systems have focused on specific systems or environments, thereby limiting the capability and application of of these systems. The majority of these proposed monitoring systems are targeted to parallel program environments where shared memory and minimal communications latency is assumed. While these proposed systems can be applied within a restricted distributed environment, such as a local area network (LAN), they do not scale into large-scale distributed systems. Although these systems have addressed monitoring intrusiveness, they neglect other design objectives, such as dynamic and scalable monitoring mechanisms.

The primary goal of this thesis was to design, develop and deploy an efficient monitoring architecture, capable of detecting primitive and composite events and of performing event correlation, within large-scale distributed system environments. The result is the Hierarchical Filtering (HiFi) monitoring system. HiFi is a highly flexible, dynamic, and scalable monitoring system, which is minimally intrusive to the application environment.

This final chapter of the thesis summarizes the approach taken to design HiFi and outlines its major design features. The impact of this work on related research and development efforts, as well as on areas outside of monitoring systems, is discussed. Lastly, an outline of potential research problems for future work and an outline of the system implementation and available documentation is provided.

9.1 Overview of the HiFi Monitoring Architecture

The architecture of HiFi is discussed in detail in Chapters 3, 4, and 5.

HiFi detects primitive and composite events through the use of dedicated monitoring processes called *Monitoring Agents (MAs)*. These agents are distributed throughout a large-scale distributed environment to facilitate event detection. Two types of MAs exist in the HiFi system: local and domain. A *local monitoring agent (LMA)* is responsible for detecting primitive events generated by applications running in the same machine as the LMA. A *domain monitoring agent (DMA)* is responsible for detecting composite events, which are beyond the scope of LMAs. When an LMA detects a primitive event, it notifies its DMA. Communication between LMAs and DMAs is hierarchical in nature and occurs to perform event correlation.

The monitoring process is started when a consumer sends a *filter program* describing a monitoring request to the subscription component in the manager program. The Filter components, which include composite events, an event expression and a filter expression, are validated and decomposed into subfilters (e.g., F1,..,Fn). This process is performed using *decomposition algorithms* so that each subfilter represents a primitive event. Based on environment specifications, including event sources and application distribution, each subfilter is then assigned to one or more LMAs using *allocation algorithms*. These decomposition and allocation algorithms are described in Chapter 4.

The monitoring system will also determine the proper DMAs to be used for evaluating the event and for evaluating the filter program's filter expression. Additionally, when MAs receive delegated monitoring tasks (e.g., subfilters) [30], they reconfigure themselves by inserting the subfilter into the filtering internal representation which is a direct acyclic graph (DAG) or a Petri Net (PN) for LMAs and DMAs, respectively [3].

HiFi implements *dynamic signaling* to suppress events at the reporting level. Dynamic signaling uses Event Reporting Stub (ERS) to identify active events and to generate notifications accordingly. This forms three types of event filtering within HiFi: identity-based, content-based, and correlation-based. MA's perform these filtering activities as dictated by the management protocol described in Section 4.2. The management protocol also removes burden from the user by automating the process of instrumentation and by administering the monitoring agents.

HiFi also provides a high-level declarative monitoring language that consists of four specifications: event specification (HESL), filter specification (HFSL), action specification (HASL) and environment specification (ESL). These specifications permit users to dynamically add, delete and modify subscriptions at run-time. Subscription consistency and synchronization within agent groups is assured by the *subscription protocol* described in Section 4.3.4. A multi-layer, multi-threaded architecture permits the LMAs and DMAs to optimize processing and space utilization through a variety of techniques outlined in Chapter 5.

9.2 System Design Objectives

While developing the HiFi monitoring system, an approach was devised by which problems are abstracted and domain requirements are fully analyzed. This approach proved effective in ensuring that HiFi comprehensively addressed system goals. The final monitoring system design was developed after studying and analyzing the target application domain and then developing a general solution framework. The design architecture, its objectives and an implementation of the design, which includes illustrations of how the design achieves major design goals, are also described. This section lists the characteristics of the HiFi system architecture and its design objectives.

Scalability: A distinguishing feature of the HiFi monitoring architecture is its scalability, in terms of the number of event producers and consumers. This is achieved through several design considerations, including (1) hierarchical filtering-based mechanisms that permit creation of additional LMAs and DMAs based on workload demands, (2) distribution of filtering workload via decomposition and allocation of monitoring tasks, (3) employment of an efficient filtering composition technique, (4) employment of space optimization techniques, and (5) use of dynamic multicasting for group communication and information dissemination. Using the priority-based monitoring mechanism, HiFi also scales to different service classes that have different time-constraints. The simulation results presented in Chapter 6 show 29% to 37% (on average) improvement of the hierarchical monitoring mean response time over centralized and decentralized monitoring architectures, respectively, with the increase of event producers from 50 to 1000 producers. It also shows that

mean response time of hierarchical monitoring is improved by 33% to 45% over centralized and decentralized monitoring architectures, respectively, with the increase of event frequency from 1% to 35%. These results indicate the high scalability of HiFi monitoring system under large number of producers and high event rate.

High-performance: The monitoring architecture supports a number of techniques to reduce monitoring latency. By reducing monitoring latency, overall system performance is increased. These techniques include: (1) a distributed hierarchical architecture that alleviates performance bottlenecks and increases concurrency in the monitoring/filtering process, (2) an efficient filter decomposition and allocation process that enables fine-grain decomposition and distribution of monitoring information, (3) a "short-cut" enhancement in the management protocol that permits the monitoring agent to avoid or bypass unnecessary communication, (4) a multi-threaded multi-layer monitoring agent architecture that increases parallelism and operation overlapping, (5) dynamic monitoring load adaptation, (6) support for several filtering optimization and implementation techniques to minimize the number of comparisons needed to correlate events, and (7) priority-based processing to permit users to attain a low latency for specific events. The throughput benchmarking shows the viability of the DAG optimization technique which results in 20% improvements over traditional DAG matching algorithm used in existing event filtering [59, 63, 94].

Dynamism: With the ever-increasing complexity of managing large-scale distributed applications, dynamism has become an essential feature in the monitoring mechanism. Although many monitoring systems are currently present in the academic and industrial domains, few can truly claim to be *dynamic*.

The HiFi architecture provides a new approach to deliver dynamism by supporting: (1) modification of monitoring demands at run-time via subscription protocols, (2) a dynamic agent hierarchy that permits adaptable and reconfigurable monitoring when the monitored environment changes, (3) programmable filtering through use of the filtering model and filtering incarnation, (4) dynamic signaling that suppresses event reporting during application execution, and (5) dynamic reliable multicasting via group masking mechanisms [4].

Non-intrusiveness: Non-intrusiveness refers to ability of a monitoring system to limit the amount of overhead monitoring creates in the application environment. No monitoring operation or system can reduce this impact to an absolute zero level. However, the goal is to reduce monitoring overhead (e.g., impact to CPU or I/O operations) to the least amount possible.

In the HiFi system, monitoring intrusiveness is controlled and minimized by: (1) limiting event propagation through three-levels of filtering (identity-based, content-based and correlation-based) and through implementation of an agent hierarchy, (2) providing predictable overhead by separating the events reporting and analysis process, resulting in a very light instrumentation routine (ERS), (3) providing two reporting schemes, *Immediate* and *Delayed*, to reduce event reporting frequency, (4) preventing *direct* communication with running programs, (5) using dynamic filtering incarnation to control monitoring granularity and to minimize the number of monitoring tasks, (6) using an event-based monitoring approach which is less intrusive than a time-based approach, and (7) using multicasting communication protocols, rather than point-to-point protocols. Limiting intrusiveness may impact other functions of the monitoring system. For example, non-intrusive techniques may limit the information freshness rate, which is one of the determining factors for attaining high-performance. However, some techniques, such as reporting mode, permit users to control the trade-offs between monitoring intrusiveness and information freshness. The perturbation results show the viability of the dynamic signaling and batching techniques to significantly reduce ($> 50\%$) the perturbation of the monitoring system. It is also shown that ERS has a very minimal effect on the application perturbation which is an important key observation to produce efficient instrumentation routines. Furthermore, the average monitoring intrusiveness is considered lower than some of the reported monitoring systems such as Falcon [32] and Issos [69].

Flexibility: Flexibility equates to easy-to-use and easy-to-manage systems. Deploying HiFi in monitoring and controlling large-scale distributed systems was part of this project's objectives. For this reason, the monitoring architecture, including its language and administrative support modules, required a considerable and comprehensive effort to include flexible user interaction.

This has been accomplished in the HiFi monitoring architecture through: (1) a

simple and comprehensible monitoring language, (2) automatic agent organization, (3) automatic code instrumentation and modification, (4) dynamic subscription, and (5) reusable object-oriented filtering components. These features increase user interaction while minimizing overhead associated with operation and administration of the monitoring system.

Expressiveness: Two design features that contribute to the expressiveness, or expressive power, of the HiFi monitoring system are the monitoring model and the monitoring language. The monitoring model includes filter incarnation to generate events and manipulate filters as "actions." This enables users to define more powerful requests, thereby increasing monitoring expressiveness.

The monitoring architecture complements the model via the Monitoring System Language (MSL), which permits users to specify monitoring demands and to control the application environment. MSL integrates detection of primitive and composite events into the same framework and has a unique interface that combines a number of valuable attributes to make it a high-level and declarative language with easy-to-use and expressive features (see Section 3.3). As an example, the filter specification (Table 3.1) within MSL supports two levels of abstraction (event expression and filter expression), which manifests a rich event correlation language. Events can be correlated from within one or several producers, or from other monitoring/management tools, such as SNMP. This permits integration of system and application event correlation. In addition, MSL provides a complete interface for specifying all requirements, including environmental requirements, which is lacking in other monitoring systems.

9.3 Impact of Contributions

The impact of this work can be seen at several levels. It has provided a foundation for studying and understanding the issues and challenges faced in monitoring and managing large-scale distributed systems. It has effectively brought forth issues, namely scalability, dynamism, and manageability, that have not been addressed appropriately in system management literature or in other monitoring implementations.

This work has also presented a monitoring system which effectively addresses these issues. A wide deployment of HiFi will demonstrate its effectiveness in monitoring a wide

variety of large-scale distributed systems and will prove its ability to improve the overall quality of service for such systems. The full impact can only be measured after HiFi has been provided to the public research community for testing and experimentation. When researchers use HiFi and fully understand its merit and limitations, we hope that new systems will be built to improve and advance this technology.

9.4 HiFi Beyond Distributed Monitoring

Event filtering is an important mechanism for a variety of application domains, including *communication protocols, distributed systems and active databases. Advances in event* filtering (design, development and optimization advances) may significantly impact these application domains.

The event filtering component incorporated in the HiFi monitoring system can be reused for building and supporting applications outside the monitoring system. Specific examples include news information dissemination, Internet resource allocation, digital library information classification, packet demultiplexing, and E-mail management services. Section 5.5 discusses an adaptive Object-Oriented framework that can be reused efficiently for developing general-purpose event management applications [8].

9.5 Outstanding Problems and Future Work

In the following sections, several areas are identified which may offer potential improvements to the HiFi monitoring system. Additionally, research problems remaining to be addressed by the monitoring architecture are also discussed.

9.5.1 Architectural Issues

Following, a number of research areas specific to the HiFi architecture are outlined.

- **Inclusion of Heterogeneous LSD Environments.** Large-scale distributed (LSD) systems are likely to span heterogeneous platforms (such as UNIX and Windows OS), include a variety of programming languages (such as C, C++, Java, etc.), and use multiple communication protocols (such as SLIP [77], TCP [85] and reliable multicast). The HiFi architecture and its agent implementation should be adapted to

monitoring heterogeneous LSD systems. Research and development in this regard should encompass related subproblems, such as incorporating Java and CORBA [66] into the HiFi architecture.

- **Integrating SNMP/CMIP.** The proposed HiFi framework calls for integration of external monitoring tools, such as SNMP and CMIP. The current version of HiFi does not include this integration. Since external monitoring tools are useful for enterprise management, inclusion of these tools in a next version of HiFi is planned.

- **Real-time Monitoring Architecture.** While HiFi provides a mechanism for processing events based on priority, it does not support either soft or hard time constraints in the event monitoring process. Issues with integrating real-time scheduling and real-time resource allocation mechanisms, such as RTP [17, 79] and RSVP [95], must be identified and these mechanisms must be integrated into the HiFi architecture. In addition, the current monitoring priority scheme can be extended to provide a global priority among all producers.

- **Dynamic Application Changes.** Enabling dynamic application changes within the monitoring system remains to be addressed. Dynamic changes will significantly increase the flexibility and dynamism of the monitoring system and will benefit items such as host mobility, new program starts (forks), and process migration.

- **Manager Conflicts.** In the existing HiFi system, managers can work concurrently to monitor a single LSD application. However, this concurrency can lead to inconsistency because of potential conflicts between manager requests. For example, different managers may require different priority levels for the same event. While each manager receives notification whenever their requests are changed, the possibility of inconsistency remains to be addressed.

- **Other Filtering Optimization.** Development of additional techniques for optimizing the filtering process is planned. One technique under consideration is based upon the matching frequency of predicates within the DAG or PN. Implementation and experimentation of this and similar techniques are needed to prove their viability to improve filtering optimization.

9.5.2 Functional Issues

A number of new functions or services can be added to the HiFi monitoring architecture to improve the quality of service. This section describes some of these functions.

- **Supporting Different Correlation Modes.** When multiple occurrences of the same event happen, the monitoring architecture currently does not address the issue of which event notification is to be considered when evaluating the filter expression. The DMA Petri Nets contains all information needed to solve this problem. However, the PN matching algorithm uses only the last occurrence to evaluate the filter expression. Permitting the user to decide how to address this issue will add more flexibility for the user.

- **Dynamic Dissemination.** Two or more managers may subscribe to the same filter (e.g., monitoring information), yet each must join its own group. This implies that the same monitoring information is forwarded over the network multiple times to different manager groups. To avoid communications overhead, managers may join groups using a group name based on the filter name. However, this may result in numerous multicast groups, which causes waste in terms of addresses and imposes overhead on some managers. This also dictates that a solution for dynamic dissemination of monitoring information be developed to achieve an optimal balance between the number of group names used (multicast addresses) and the number of messages sent to managers. Further investigation and analysis of this problem is part of the future research plan.

- **Fault Tolerance.** A fault tolerant monitoring architecture, to include its agent and manager entities, is required to offer a reliable service for LSD environments. The fault tolerance algorithm and related techniques should leverage existing reliability features, such as multi-point communications and failure notification propagation, to build a fully fault tolerant architecture.

- **Supporting Event Ordering.** Event ordering (partial or total) is an important service in many monitoring applications [48]. For example, event traces may be merged or combined based on casual ordering. Clock synchronization (e.g., using

logical and vector clock algorithms) can be integrated into the current architecture to provide more robust event ordering services.

The current architecture assumes that users may use NTP [61] and may incorporate NTP information in event notifications to achieve physical clock synchronization. In addition, consumers can also rely on the total ordering service supported by RMP [92] to provide logical event ordering. However, an event ordering mechanism must be implicitly supported within the monitoring architecture itself to avoid user intervention.

9.5.3 Application Issues

By providing explicit application support, the HiFi monitoring system can achieve a larger deployment and can be useful to more environments. For this reason, plans exist to extend the architecture to directly support the following applications:

- **Performance Measurement for Internet Services.** HiFi can be extended to provide Internet management services, such as providing performance measurements for Internet Service Providers (ISP) and QoS management for application steering and tuning of Internet services [89].

- **Additional Distributed Debugging Features.** In addition to error correlation and event traces, other distributed debugging issues need to be explored and implemented in the HiFi system. These include: distributed break points, instant reply, and integrating gdb (GNU debugger) for getting and setting variables in remote programs.

- **Visualizing Monitoring Results.** Visualization of monitoring results is envisioned as one of several additional applications to be integrated into the monitoring language. Visualization will increase user flexibility in using and analyzing monitoring information.

9.5.4 Language Issues

In the following section, a number of research areas designed to improve the usability and expressiveness of the monitoring language are presented.

- **Time Support and Event Temporal Operators.** Time functions and temporal event ordering operators, such as *Before* and *After*, will be added to the monitoring language to increase its expressive power. These functions are directly related to, and will facilitate resolution of, the event ordering issue discussed in Section 9.5.2.

- **More Powerful Abstraction.** Although the monitoring language is high-level and declarative, additional abstraction can be attained by utilizing the current language with a low-level interpretation. The high-level abstraction language should be target oriented, which will require users to only define the monitoring target (goal). Corresponding filter specifications will be generated automatically. In other words, users can specify the ultimate monitoring target without having to specify intermediate monitoring tasks. Users, therefore, would not need to check the validity of event correlation in the filter program.

9.6 Status and Availability

Two generations, or versions, of HiFi have been released for experimentation and implementation. Version 0.6b was developed in December 1998 and only provides support for the main language constructs and for basic functions of the monitoring system. The more recent version, HiFi 1.0b, was extended to support more advanced features, such as flexible instrumentation, special language constructs (e.g., `ANY`, `TRUE` and `ALL`), automatic agent organization and most of the dynamic monitoring features.

HiFi 1.0b is the version used for monitoring the IRI virtual classroom as described in Section 7.1. Source code is publicly available from `http://www.cs.odu.edu/~ehab/Projects/HiFi`. Technical reports and papers [2, 3, 5, 6, 7, 8] are available in the technical report archive at the Computer Science Department of the Old Dominion University. For source code compilation, the following packages are required:

- Solaris 2.5 or Sun OS 5.5 or higher
- GNU g++ compiler (version 2.7.0)
- flex Lexical Analyzer (version 2.5.4)
- Yacc parser (version 2.0)

- Commonly used C++ list class templates
- Reliable Multicast Protocol (RMP)
- Dynamic Reliable Multicast Service (RMS 2.5)

REFERENCES

[1] H. M. Abdel-Wahab and M. Feit, "Xtv: A framework for sharing x window clients in remote synchronous collaboration," in *Proc. IEEE TriComm '91: Communications for Distributed Applications & Systems*, pp. 159–167, April 1991.

[2] E. Al-Shaer, "High-performance event filtering: Survey and evaluation, TR-96-12," Tech. Rep., Computer Science Department, Old Dominion University, August 1996.

[3] E. Al-Shaer, "Event Filtering Framework: Key Criteria and Design Trade-offs," in *IEEE* 21^{st} *Int. Conference on Computer Software and Application*, pp. 84–89, August 1997.

[4] E. Al-Shaer, H. Abdel-Wahab, and K. Maly, "Application-Layer Group Communication Server for Extending Reliable Multicast Protocols Services," in *IEEE Int. Conference on Network Protocols*, pp. 267–274, October 1997.

[5] E. Al-Shaer, H. Abdel-Wahab, and K. Maly, "Hierarchical filtering-based monitoring architecture for large-scale distributed systems," in *Proc. Int. Conference on Parallel and Distributed Computing Systems*, pp. 422–427, October 1997.

[6] E. Al-Shaer, H. Abdel-Wahab, and K. Maly, "A scalable monitoring architecture for managing distributed multimedia systems," in *Proc. IFIP/IEEE Int. Conference on Managing Multimedia Networks and Services*, pp. 237–248, July 1997.

[7] E. Al-Shaer, H. Abdel-Wahab, and K. Maly, "High-performance Monitoring Architecture for Large-scale Distributed Systems Using Event Filtering," in *CS&I'97: Third International Conference on Computer Science & Informatica*, vol. 3, pp. 42–46, March 1997.

[8] E. Al-Shaer, M. Fayad, H. Abdel-Wahab, and K. Maly, "Adaptive Object-Oriented Filtering Framework for Event Management Applications," *To appear in ACM Computing Survey*, December 1998.

[9] E. Al-Shaer, A. Youssef, H. Abdel-Wahab, K. Maly, , and C. Overstreet, "Reliability,

Scalability and Robustness Issues in IRI," in *IEEE Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'97)*, June 1997.

[10] S. Alexander, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie, "High Speed and Robust Event Correlation," *IEEE Communication Magazine*, pp. 433–450, May 1996.

[11] F. Baiardi and N. D. G. Vaglini, "Development of a Debugger for Concurrent Language," *IEEE Transactions on Software Engineering*, pp. 547–553, April 1986.

[12] M. L. Bailey, B. Gopal, M. A. Pagels, L. Peterson, and P. Sarkar, "PathFinder: A Pattern-Based Packet Classifier," in *Proc. 1st Symposium on Operating System Design and Implementation*, pp. 24–42, USENIX Association, November 1994.

[13] P. Bates, "Debugging Heterogeneous Distributed Systems Using Event-based Models Behavior," in *Proc. Workshop in Parallel and Distributed Systems Debugging*, pp. 11–22, 1988.

[14] K. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit.* Los Alamitos: IEEE Computer Society Press, 1994.

[15] R. Boutaba and S. Znaty, "Architectural Approach for Integrated Network and Systems Management," *Computer Communication Review*, vol. 25, no. 5, pp. 13–38, 1995.

[16] R. T. Braden, "A Pseudo-Machine for Packet Monitoring and Statistics," in *Proc. the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pp. 200–209, ACM, August 1988.

[17] I. Busse, B. Deffner, , and H. Schulzrinne, "Dynamic QoS Control of Multimedia Applications based on RTP," *Second Workshop on Protocols for Multimedia Systems*, October 1995.

[18] J. Case, K. McCloghrie, M. Rose and S. Waldbusser, "Introduction to SNMPv2," RFC1441, Network Working Group, IETF, April 1993.

[19] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data and Knowledge Engineering*, vol. 14, pp. 1–26, November 1994.

[20] P. S. Chen, "The entity-relation model–toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, pp. 9–36, March 1976.

[21] R. S. Dodd and C. V. Ravishankar, "Monitoring and debugging distributed real-time programs," *Software–Practice and Experience*, vol. 22, pp. 863–877, 1992.

[22] L. Ehley, B. Furth, and M. Ilyas, "Evaluation of Multimedia Synchronization Techniques ," in *Proc. International Conference of Multimedia Computing Systems*, pp. 110–119, May 1994.

[23] C. Fischer and R. L. Jr., *Craft A Compiler with C.* Redwood, CA: Benjamin/Cummings, 1995.

[24] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," pp. 342–356, October 1995.

[25] V. Frost and B. Melamed, "Traffic Modeling for Telecommunication Networks," *IEEE Communication Magazine*, vol. 32, pp. 70–80, March 1994.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software Architecture.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1995.

[27] S. Gatziu and K. R. Dittrich, "Detecting Composite Events in Active Database Systems Using Petri Nets," in *Proc.* 4^{th} *International Workshop on Research Issues in Data Engineering: Active Database Systems*, pp. 2–9, February 1994.

[28] N. Gehani, H. V. Jagadish, and O. Shmueli, "Event Specification in an Object-Oriented Database," in *Proc. ACM SIGMOD International Conference on Management of Data*, Lecture Notes Computer Science, 1992.

[29] N. Gehani, H. V. Jagadish, and O. Shmueli, "COMPOSE A System for Composite Event Specification and Detection," in *Book Chapter in Advanced Database Concepts and Research Issues*, pp. 81–90, Lecture Notes Computer Science, 1993.

[30] S. Y. German Goldszmidt and Y. Yemini, "Network Management by Delegation - the MAD approach," in *Proc. 1991 CAS Conference*, pp. 347–359, 1991.

[31] M. Ginsberg, *Essentials of Artificial Intelligence.* New York, NY: Morgan Kaufmann, 1993.

[32] W. Gu, G. Eisenhauer, E. Kraemer, K. Stasko, J. Vetter, and N. Mallavarupu, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," in *Proc. FRONTIERS'95*, pp. 11–19, February 1995.

[33] P. K. Harter, D. M. Heimbigner Jr., and R. King, "IDD:Am Interactive Distributed Debugger," in *Proc. International Conference on Distributed Computing Systems*, pp. 498–506, 1985.

[34] J. Haugdahl, "Benchmarking LAN protocol analyzers," in *Proc.* 13^{th} *IEEE Conference on Local Computer Networks*, October, 1988.

[35] B. Helmbold and D. Luckham, "Debugging Ada Tasking Program," *IEEE Software*, pp. 47–57, 1985.

[36] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle, "Distributed Performance Monitoring: Methods, Tools and Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 585–597, June 1994.

[37] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation.* Reading, Massachusetts: Addison-Wesley, 1979.

[38] A. Hough and J. Cuny, "Belvedere: Prototype of a Pattern-oriented Debugger for Highly Parallel Computation," in *Proc. International Conference on Parallel Processing*, pp. 735–738, 1987.

[39] Isis Distributed Systems, Inc., Marlboro, MA, *Isis Users's Guide: Reliable Distributed Objects for C++*, April 1994.

[40] V. Jacobson, C. Leres, and McCanne, "The Tcpdump Manual Page," Lawrence Berkeley Laboratery, June 1989.

[41] R. Jain, *The Arts of Computer Systems Performance Analysis.* Reading, Massachusetts: Addison-Wesley, 1991.

[42] R. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June 1988.

[43] J. F. Jordann and M. E. Paterok, "Event Correlation in Heterogeneous Networks Using the OSI Management Framework," in *Proc.* 1^{st} *International Symposium on Integrated Network Management*, pp. 365–379, IFIP, 1989.

[44] J. Joyce, G. Lomow, K. Slind and B. Unger, "Monitoring Distributed Systems," *ACM Transactions on Computer Systems*, vol. 5, no. 2, pp. 121–50, 1987.

[45] K. S. Klemba, "Openview's Architectural Models," in *Proc.* 1^{st} *International Symposium on Integrated Network Management* (B. Meandzija and J. Westcott, eds.), pp. 565–572, IFIP, 1989.

[46] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, "Using Hy^{+} for Network Management and Distributed Debugging," in *Proc. Cascon*, pp. 450–471, October 1993.

[47] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, "Coding Approach to Event Correlation," in *Proc. Fourth International Symposium on Integrated Network Management*, May 1995.

[48] L. Lamport, "Time, Clocks and Ordering of Events in a Distributed System," *Communication of ACM*, vol. 21, no. 7, pp. 558–565, 1987.

[49] B. Lazzerini and C. A. Prete, "Disdeb: An Interactive high-level Debugging System for a Multi-microprocessor System," *Microprocessor, Microprogram*, vol. 18, pp. 401–408, 1986.

[50] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Reply," *IEEE Transactions on Computers*, vol. 36, pp. 471–482, April 1987.

[51] J. Levine, T. Mason, and D. Brown, *Lex & Yacc.* Sebastopol, CA: O'Reilly & Associates, 1995.

[52] B. Lewis and D. J. Berg, eds., *Threads Primer: A Guide to Multithreaded Programming.* Sun Soft Press, 1996.

[53] B. J. MacLennan, *Principles of Programming Languages.* San Diego, CA: Sauders Collage Publishing, 1987.

[54] A. D. Maio, S. Ceri, and S. Reghizzi, "Execution Monitoring and Debugging Tool for Data Using Relational Algebra," in *Proc. Ada International Conference*, 1985.

[55] A. D. Malony, D. A. Reed, and H. A. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 433–450, July 1992.

[56] K. Maly, H. Abdel-Wahab, C. M. Overstreet, C. Wild, A. Gupta, A. Youssef, E. Stoica, and E. Al-Shaer, "Interactive distance learning over intranets," *IEEE Internet Computing*, vol. 1, pp. 60–71, February 1997.

[57] M. Mansouri, *Monitoring Distributed Systems*, Reading, Massachusetts: Addison-Wesley, 1994.

[58] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman, "Tools for distributed Application Management," *IEEE Computer*, vol. 24, pp. 42–51, August 1991.

[59] S. McCanne and V. Jacobson, "The BSD Packet Filter," in *Winter USENIX*, pp. 259–269, USENIX Association, January 1993.

[60] C. E. McDowell and D. D. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, vol. 21, pp. 593–622, December 1989.

[61] D. Mills, "Simple network time protocol (SNTP) version 4 for ipv4, IPv6 and OSI," RFC 2030, Network Working Group, IETF, October 1996.

[62] J. C. Mogul, "Efficient Use of Workstations For Passive Monitoring of Local Area Networks," in *Proc. Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1990.

[63] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The Packet Filter: An Efficient Mechanism of User-Level Network Code," in *Proc.* 11^{th} *Symposium on Operating Systems Principles*, pp. 39–51, ACM, November 1987.

[64] J. Mott, A. Kandel, and T. Baker, *Discrete Mathematics for Computer Scientists and Mathematicians.* Englewood Cliffs, NJ: Prentice Hall, 1987.

[65] B. Neil and J. J. Garcia-Luna-Aceves, "Improving Internet Multicast with Routing Labels," pp. 241–250, October 1997.

[66] Object Management Group, *The Common Object Request Broker: Architecture and Specification.* Tech. Rep. CCITT X.734, 1993.

[67] Object Management Group, *The Common Object Request Broker: Event Service Specification.* Tech. Rep. CCITT X.734, 1993.

[68] D. Ohsie, and S. Kliger, "Network Event Management Surveys," Tech. Rep., System Management Arts (SMARTS), April 1993.

[69] D. Olge, K. Schwan, and R. Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 593–622, December 1989.

[70] F. G. Pagen, *Formal Specification of Programming Languages.* Englewood Cliffs, NJ: Prentice Hall, 1981.

[71] S. Pejhan, A. Eleftheriadis, and D. Anastassiou, "Refinements to Rate-Based Congestion Control with Extension to Multipoint, Multimedia Applications," *IEEE/ACM Transactions on Networking*, vol. 4, pp. 121–50, June 1996.

[72] G. Perrow, "Monitoring Techniques in Distributed System Management, Technical Report 421," Tech. Rep., University of Western Ontario, March 1994.

[73] Rational Software, "Application Note: Doubling the Performance of Xman Using Quantify," Tech. Rep., http://www.rational.com/products/quantify, 1998.

[74] T. Risch, "Monitoring Database Objects," in *Proc. VLDB*, August 1989.

[75] M. T. Rose, *The Simple Book: An Introduction to Internet Management.* Englewood Cliffs, NJ: Prentice Hall, April 1994.

[76] H. A. Schmid, "Systematic framework design by generalization," *Communication of ACM*, vol. 40, pp. 48–51, October 1997.

[77] K. Schneider and S. Venters, "PPP Serial Data Transport Protocol (SDTP)," RFC 1963, Network Working Group, IETF, August 1996.

[78] B. Schroeder, "On-line Monitoring: A Tutorial," *IEEE Computer*, vol. 28, pp. 72–78, June 1995.

[79] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications. Audio-Video," RFC 1889, Transport Working Group, IETF, January 1996.

[80] K. R. Sheers, "HP OpenView Event Correlation Service," Tech. Rep., Hewlett-Packard Journal, 1996.

[81] M. Sloman, ed., *Network and Distributed System Management.* Reading, Massachusetts: Addison-Wesley, 1994.

[82] W. Stallings, ed., *SNMP, SNMPv2 and CMIP: The Practical Guide to Network Management Standards.* Reading, Massachusetts: Addison-Wesley, 1993.

[83] I.S.O. Standarization, *Information Processing Systems - Open Systems Interconnection - Part 5: Event Report Management Function.* Tech. Rep. CCITT X.734, 1993.

[84] W. R. Stevens, *Advanced Programming in the UNIX Environment.* Reading, Massachusetts: Addison-Wesley, 1993.

[85] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols.* Reading, Massachusetts: Addison-Wesley, 1994.

[86] W. R. Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP and the UNIX Domain Protocols.* Reading, Massachusetts: Addison-Wesley, 1996.

[87] Sun Microsystems Inc., "The Snoop(1M) Manual Page," in *SunOS 5.5 Reference Manual*, January 1995.

[88] A. S. Tanenbaum, *Modern Operating Systems.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[89] Tivoil, Inc., http://www.tivoli.com/o_products/html/xsite_mia_wp.html, *Managing Internet Applications.*

[90] Tivoil, Inc., http://www.tivoli.com/o_products/html/dm_ds.html, *Tivoli Distributed Monitoring.*

[91] C. Wang and M. Schwartz, "Fault detection with multiple observers," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 48–55, February 1993.

[92] B. Whetten, T. Montgomery, and S. Kaplan, "A High Performance Totally Ordered Multicast Protocol," in Proc.*Theory and Practice in Distributed Systems*, pp. 33–57, 1994.

[93] O. Wolfson, S. Sengupta, and Y. Yemini, "Managing Communication Networks by Monitoring Databases," *IEEE Transactions on Software Engineering*, pp. 944–953, September 1991.

[94] M. Yuhara, B. Bershad, C. Maeda, and J. E. B. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," in *Proc. Winter 1994 USENIX Conference*, USENIX Association, January 1994.

[95] L. Zhang, S. Deering, D. Estrin, S. Shenker, , and D. Zappala, "RSVP: A new resource ReSerVation Protocol," *IEEE Network*, Sept. 1993.

APPENDIX A

DISTRIBUTED "HELLO WORLD" MONITORING EXAMPLE

This appendix illustrates by step-by-step example how HiFi can be used to instrument and monitor distributed "HelloWorld" application described in Chapter 6. The HelloWorld distributed programs generate randomly two types of events: *HelloEvent* and *WorldEvent* where each one indicates the event type (in `Info`, the machine name (in `Machine`) and the the sequence number or time stamp (`TStamp`) of this event. A user wants to know if any at least two HelloWorld distributed programs send "Hello" (*HelloEvent*) and "World"(*WroldEvent*) simultaneously or, in other words, having the same sequence number `TStamp`. The following are the steps followed to perform this monitoring task:

(1) The user initialize the program code for instrumentation by including "ERSvar.h" in the headers, initializing ERS using `ERSInit`, and inserting user sensors (`ReportEvent`). The code after this preparation is show below:

A.1 HelloWorld.cc

```
#include<stdio.h>
#include"ERSvar.h"
#include"./distribution/random_variates.h"
#define  NapTime 1 /* sec*/
extern int ReportEvent(char *Ev,char *Mod,char* Fun,char *Rep, int cnt, ...);
extern int ConnectToLMA(void);
```

```
extern int ERSInit(int);

main(int argc, char **argv) {
  int sam,TStamp=1;
  char *ModName = new char[16], *FuncName= new char [32];
  char *Info = new char[64], *Machine= new char [32], *Type = new char[16];
  GetMachineName(Machine); strcpy(FuncName,"Greetings");
  ERSInit(AUTOMATIC);   /* initialization ERS */
  for (;;)  {
    Bernoulli *ber = new Bernoulli(0.5); /* either 1 or 0 */
    sam = ber->sample();
    if (sam ==1)    {
      printf("HELLO = %d\n", TStamp);
      ReportEvent("HelloEvent");
    }
    if (sam ==0)    {
      printf("WORLD = %d\n", TStamp);
      ReportEvent("WorldEvent");
    }
    sleep(NapTime);
    TStamp++;
  } // end of for (;;)
} // end of main
```

(2) The user writes in a file the environment, events, filter specifications conforming to ESL, HESL, HFSL, respectively. Let us call this file: "ex_helloworld" which is shown below:

A.2 Language Script: "ex_helloworld"

Environment Specs

```
ODU = dragon, elf;
VB = naga, zeus@
VA_State = ODU, VB;
USA = VA_State &
HelloWorld = *.
```

Events Specs

```
EVENT= {ModuleName=HelloWorld, FuncName=Greetings, Immediate;
Machine="ANY", Info="Hello", TStamp=ANY} HelloEvent.
EVENT= {ModuleName=HelloWorld, FuncName=Greetings, Immediate;
Machine="ANY", Info="World", TStamp=ANY} WorldEvent.
```

Filters Specs

```
FILTER= [(HelloEvent && WorldEvent)];
        [(HelloEvent.TStamp = WorldEvent.TStamp &&
          HelloEvent.Machine != WorldEvent.Machine)];
        [FORWARD]; HelloEventFilter.
```

(3) The user starts the instrumentation process by inputing the language script file, "ex_helloworld" to *MLI* program using *-i* option to initiate the instrumentation operation. The partial output script is shown below:

A.3 Partial Output

```
zues:/home/ehab/HiFi>MLI -i -a ex_helloworld
HiFi: Hierarchical Filtering-based Monitoring System -Version 0.1b
MLI:Monitoring Language Interface Version 0.3b

Language Spec File=ex_helloworld, Output File=MLI.out
Constructing Monitoring-based Information .....
.....................
... Deleted Stuff ..
.....................
End of Parsing! See the Output Files "MLI.out"

Enter File Name > HellowWorld.cc
File name=HelloWorld, extension=cc
<<< Starting the Instrumentation Precess >>>
Generating .HiFiHelloWorld.cc .....
Reading ''Makefile'' and generating Makefile.HiFi .....
<<<. Instrumentation is Done >>>
```

```
Compiling and Linking with ERS object: ''make -f Makefile.HiFi'' (Y\N) > y
... compiling finished ...
YOU CAN START NOW: YOUR PROGRAM WILL START THE AGENTS
   USE MLI -a <filename> TO OPERATE THE MANAGER
 FASTEN YOUR SEAT BELT AND DRIVE SAFELY with HiFi!
```

(4) The user's sensors are replaced, now, by the extended "system sensors" as shown below. The instrumentation process generates a new instrumented file, *.HiFiHelloWorld.cc*, which is typical the same as the original file, *HelloWorld.cc*, but the user's sensors are replaced with extended "system sensors" as shown below. The "Makefile" file is also modified accordingly.

A.4 System Sensors

```
ReportEvent("HelloEvent","HelloWorld","Greetings","IMMEDIATE",3,
     "Machine",STRING,Machine,"Info",STRING,"Hello","TStamp",INTEGER,TStamp);
ReportEvent("WorldEvent","HelloWorld","Greetings","IMMEDIATE",3,
     "Machine",STRING,Machine,"Info",STRING,"Hello","TStamp",INTEGER,TStamp);
```

(5) Now, the user should starts the manager program, and then starts the application programs (instrumented) in the machines specified in the environment specifications, but without **-i** option as shown below (**-a** means ESL, HESL and HFSL are all in the same file) :

```
zues:/home/ehab/HiFi>MLI -a ex_helloworld
....
dragon:/home/ehab/HiFi>HelloWorld
....
elf:/home/ehab/HiFi>HelloWorld
....
zues:/home/ehab/HiFi>HelloWorld
....
naga:/home/ehab/HiFi>HelloWorld
....
```

Notice that "HelloWorld" *must* run from the same file system (may be directory) that contains LMA and DMA programs.

APPENDIX B

MONITORING-KNOWLEDGE BASE

B.1 Environment Tables

```
class ItemInfo {
  int         num;
  String      name;
  ItemInfo() { }
  ItemInfo(int nu, String nm) : num(nu), name(nm){ }
  int operator!=( const ItemInfo & Rhs) const    {
    if (Rhs.num == -1)         return (name != Rhs.name);
    else if (Rhs.name == "*")  return (num != Rhs.num);
    else                       return ((name != Rhs.name) && (num != Rhs.num));
  }
};
template<class Rtype> class EnvTable {
  int               num;
  String            name;
  List<Rtype>       *asslist;
  EnvTable() { }
  EnvTable(int nu, String nm, List<Rtype> *Lin) : num(nu), name(nm), asslist(Lin){ }
  int operator!=( const EnvTable & Rhs) const    {
    if (Rhs.num == -1)           return (name != Rhs.name);
    else if (Rhs.name == "*")    return (num != Rhs.num);
    else                         return ((name != Rhs.name) && (num != Rhs.num));
  }
};
```

```
class HierarchyInfo {
  int         level;
  String      domname;
  HierarchyInfo() { }
  HierarchyInfo(int level, String domname) : level(level), domname(domname){ }
  int operator!=( const HierarchyInfo & Rhs) const    {
    if (Rhs.level == -1)             return (domname != Rhs.domname);
    else if (Rhs.domname == "*")     return (level != Rhs.level);
    else                 return ((domname != Rhs.domname) || (level != Rhs.level));
  }
};

List< EnvTable<ItemInfo> >      DomainToMacTable;
ListItr< EnvTable<ItemInfo> >   PtrDomainToMacTable(DomainToMacTable);
List< EnvTable<ItemInfo> >      MacToDomainTable;
ListItr< EnvTable<ItemInfo> >   PtrMacToDomainTable(MacToDomainTable);
List< EnvTable<ItemInfo> >      SuperdomainTable;
ListItr< EnvTable<ItemInfo> >   PtrSuperdomainTable(SuperdomainTable);
List< EnvTable<ItemInfo> >      ModuleToLocTable;
ListItr< EnvTable<ItemInfo> >   PtrModuleToLocTable(ModuleToLocTable);
List<HierarchyInfo>         HierarchyInfoList;
ListItr<HierarchyInfo>      PtrHierarchyInfoList(HierarchyInfoList);
```

B.2 Primitive Events Tables

```
class ATTRIBUTE {
  String      name;
  int         rel;  /* < > = .. */
  Value       val;

  ATTRIBUTE() { }
  ATTRIBUTE(String nm, int r, Value v) : name(nm), rel(r), val(v){ }
  int operator!=( const ATTRIBUTE & Rhs) const    {
    return ((name != Rhs.name) && (rel != Rhs.rel));
  }
};
class PrimEvent {
  int               id;
  int               attctr; /* from 0 -> N-1 */
  String            name;
  String            ModName;
  String            FuncName;
  String            ReportMode;
  List<ATTRIBUTE>   *varatt;
  PrimEvent() { }
  PrimEvent(int nu,int ctr, String nm,
    String mod, String fn, String rep, List<ATTRIBUTE> *Latt) :
     id(nu), attctr(ctr), name(nm), ModName(mod), FuncName(fn),
     ReportMode(rep),varatt(Latt){ }
  int operator!=( const PrimEvent & Rhs) const {
    if (Rhs.id == -1)          return (name != Rhs.name);
    else if (Rhs.name == "*")  return (id != Rhs.id);
    else         return ((name != Rhs.name) && (id != Rhs.id));
  }
};
List<PrimEvent>      PrimEventTable;
ListItr<PrimEvent>   PtrPrimEventTable(PrimEventTable);
```

B.3 Composite Events Tables

```
class CompEventBody {
  int                  scope;
  String               name;
  int                  rel; /* between this event and next one */
  int                  type;
  CompEventBody() { }
  CompEventBody(int sc,String nm,int r,int t) :
     scope(sc), name(nm), rel(r), type(t){ }
  int operator!=( const CompEventBody & Rhs) const    {
    if (Rhs.scope == -1)
       return (name != Rhs.name);
    else if (Rhs.name == "*")
       return (scope != Rhs.scope);
    else
       return ((name != Rhs.name) && (scope != Rhs.scope));
  }
};
class CompEvent {
  int                  id;
  int                  max_scope;
  String               name;
  List<CompEventBody>  *evlist;
  CompEvent() { }
  CompEvent(int nu, int max, String nm, List<CompEventBody> *Lev) :
     id(nu), max_scope(max), name(nm), evlist(Lev){ }
  int operator!=( const CompEvent & Rhs) const    {
    if (Rhs.id == -1)
       return (name != Rhs.name);
    else if (Rhs.name == "*")
       return (id != Rhs.id);
    else
       return ((name != Rhs.name) && (id != Rhs.id));
  }
};
List<CompEvent>      CompEventTable;
ListItr<CompEvent>   PtrCompEventTable(CompEventTable);
```

B.4 Filter Tables

```
class FilerExpr {
  String          attname;
  String          evname;
  int             rel;
  Value           *value;
  List<ItemInfo>  *LMAs; /* machine names */
  List<ItemInfo>  *DMAs; /* domain names */
  FilterExpr() { }
  FilterExpr(String an, String en, int r, Value *v, List<ItemInfo> *lm,
     List<ItemInfo> *dm) :
  attname(an), evname(en), rel(r), value(v), LMAs(lm), DMAs(dm){ }
  int operator!=( const FilterExpr & Rhs) const
     return (attname != Rhs.attname);
};
class EventExpr {
  int             id;
  String          evname;
  int              scope;
  int             rel;
  List<ItemInfo>  *LMAs; /* LMA names */
  List<ItemInfo>  *DMAs; /* DMA names */
  EventExpr() { }
  EventExpr(int id,String nm, int scope, int rel,List<ItemInfo> *LMAs,
    List<ItemInfo> *DMAs):
  id(id),evname(nm),scope(scope),rel(rel), LMAs(LMAs), DMAs(DMAs){ }
  int operator!=(const EventExpr & Rhs) const  {
    if (Rhs.id == -1) return (evname != Rhs.evname);
    else if (Rhs.evname == "*")  return (id != Rhs.id);
    else  return ((evname != Rhs.evname) && (id != Rhs.id));
  }
};
```

```
class Filter {
  int              id;
  String           fname;
  String           action;
  List<EventExpr>  *EX;  /* has all events invovled */
  List<FilterExpr> *FX;  /* has all attributes */
  Filter() { }
  Filter(int id,String fname,String action,List<EventExpr> *EX,
         List<FilterExpr> *FX):
         id(id),fname(fname),action(action),EX(EX),FX(FX){ }
  int operator!=(const Filter & Rhs) const  {
    if (Rhs.id == -1)
       return (fname != Rhs.fname);
    else if (Rhs.fname == "*")
       return (id != Rhs.id);
    else
       return ((fname != Rhs.fname) && (id != Rhs.id));
  }
};
List<Filter>     FilterTable;
ListItr<Filter>  PtrFilterTable(FilterTable);
```

APPENDIX C

CLASSES AND ALGORITHMS OF DAG AND PN

C.1 DAG Classes and Iterators

```
class Value {
  int       rel;  /* < > = .. */
  Item      item;
  Value() { }
  Value(int r, Item it) : rel(r), item(it) { }
  int operator!=( const Value & Rhs) const    {
    return (rel != Rhs.rel);  }
};
class ATTRIBUTE {
  String        name;
  List<Value>   *val;
  ATTRIBUTE() { }
  ATTRIBUTE(String nm, List<Value> *v) : name(nm), val(v){ }
  int operator!=( const ATTRIBUTE & Rhs) const    {
    return (name != Rhs.name);  }
};
class DAGNodeInfo {
  String            FuncName;
  String            ReportMode;
  List<ATTRIBUTE>   *Pred;
  DAGNodeInfo() { }
  DAGNodeInfo(String fn, String rep, List<ATTRIBUTE> *Latt) :
     FuncName(fn), ReportMode(rep), Pred(Latt){ }
  int operator!=( const DAGNodeInfo  & Rhs) const  {
```

```
    return (FuncName != Rhs.FuncName);  }
};
class DAGNode {
  String              ModName;
  List<DAGNodeInfo>   *DAGNodeInfoTable;
  DAGNode() { }
  DAGNode(String mn, List<DAGNodeInfo> *DN): ModName(mn), DAGNodeInfoTable(DN){ }
  int operator!=( const DAGNode  & Rhs) const    {
    return (ModName != Rhs.ModName);  }
};
List <DAGNode>    DAG;
ListItr <DAGNode> PtrDAG(DAG);

class DAGItr
{
 public:
  DAGItr() {}
  ~DAGItr() {}

  virtual int InsertFilter(FilterMsg *Subfilter);
  virtual int DeleteFilter(String *FilterName);
  virtual int ModifyFilter(String *FilterName, FilterMsg *Filter);
  virtual String DAGMatchEvent(FilterMsg *event);
  virtual int PredEvaluate (FMsgPred *event, int rel, Item *dag);
};

struct Item {
  int         type; /* 1 int, 2 float 3 string*/
  union  {
    int   intv; double  fltv; char  strv[30];
  } value;
};
```

C.2 PN Classes and Iterators

```
struct RValue {
  union  {
    int      intv;
    double   fltv;
    char     strv[MAX_LENGTH];
  } item;
};
class Place {
  int                 mark;
  int                 mode; /* 0,1,2 means consider the 1st, the last and
       event every occurance */
  String              EVname;
  List<EventState>    *OccurStates;
  Place  () { }
  Place (int mk, int md, String name, List<EventState> *es):
  mark(mk), mode(md), EVname(name), OccurStates(es) { }
  int operator!=( const Place  & Rhs) const  {
    if (EVname != Rhs.EVname)
       return (EVname != Rhs.EVname);  }
};
class Expression {
  String          Lattname;
  String          Levname;
  int             rel;
  int             Rtype; /* 1 int, 2 float 3 string*/
  String          Rattname;
  String          Revname;
  RValue         *Rvalue;
  Expression() { }
  Expression(String la,String ln,int rel,int type,String ra,String rn,RValue *val):
      Lattname(la),Levname(ln),Levid(lid),rel(rel),
      Rtype(type), Rattname(ra),Revname(rn),Revid (rid),Rvalue(val) { }
  int operator!=( const Expression  & Rhs) const  {
    if (Rhs.Levname != Levname)       return (Levname != Rhs.Levname);
```

```
    else if (Rhs.Revname != Revname)  return (Revname != Rhs.Revname);
  }
};

class PNnode {
  int               NodeID;
  int               fire_flag; /* fires if 0, otherwise it is # of places + 1 */
  String            FilName;
  String            Action;
  int               pred_count;
  List<Place>       *places;
  List<Expression>  *PNFX;
  PNnode() { }
  PNnode(int id, int mk, int f, String fn,String ac, int ct,
 List<Place> *pl, List<Expression> *fx):
  NodeID(id),default_mark(mk),fire_flag(f),FilName(fn),Action(ac),
  pred_count(ct),places(pl), PNFX(fx) { }
  int operator!=( const PNnode  & Rhs) const {
    return (NodeID != Rhs.NodeID);
  }
  List<PNnode>          PN;
  ListItr<PNnode>       PtrPN(PN);

class PNItr
{
 public:
  PNItr() {}
  ~PNItr() {}

  virtual int InsertFX(FilExprMsg *FX);
  virtual int DeleteFX(FilExprMsg *FX);
  virtual int ModifyFX(FilExprMsg *FX);
  virtual int PNMatchEvent(FilterMsg *PrimEvent);
  virtual int FXEvaluation(PNnode *FX);
  virtual int RestoreNode(PNnode *FX);
};
```

C.3 DAG Event Matching Algorithm

```
String DAGItr::DAGMatchEvent(FilterMsg *event)
{
  max=event->PredCount;
  ListItr <DAGNode> DAGFinder(DAG);
  List<DAGNodeInfo> *dumnode= new List<DAGNodeInfo>;
  if (DAGFinder.Find(DAGNode(event->mod,dumnode)))  {
    /* check here for common function names */
    ListItr <DAGNodeInfo> DAGInfoFinder(*(DAGFinder().DAGNodeInfoTable));
    List<ATTRIBUTE> *dumAtt= new List<ATTRIBUTE>;
    if (DAGInfoFinder.Find(DAGNodeInfo(event->func,event->rep,dumAtt))) {
      while (count < max)  {   /* check here for common Attribute Name */
        ListItr <ATTRIBUTE> AttFinder(*(DAGInfoFinder().Pred));
        if (AttFinder.Find(ATTRIBUTE(event->Predicates[count].name,dumVal))) {
          /* evaluate the predicate */
          for (ListItr<Value> VItr(*(AttFinder().val));+VItr;++VItr)  {
             AttFinder++;
             if (!strncmp((char*)AttFinder().name,"EVID",4))  {
                return AttFinder().name; /* DETECTED */
             }
             if (!PredEvaluate(&(event->Predicates[count]), VItr().rel,
                 &(VItr().item)))
                return "REJECTED";
             else  break;
          } /* end of for */
        }
        else  /*common attribute name not found*/
          return "REJECTED" /*event rejected */
        count++;
      } /* end of while */
    }
    else return "REJECTED"; /* common func name not found */
  } /* common module name not found */
  else  return "REJECTED";
} /* end of DAGMatchEvent() */
```

C.4 PN Event Matching Algorithm

```
int PNItr::PNMatchEvent(FilterMsg *PrimEvent) {
  List <int> *ids,*dumids = new List<int>; int  result=0;
  PtrEventsTable.Zeroth();
  if (PtrEventsTable.Find(EventNodes(PrimEvent->id,-1,dumids)))
    ids = PtrEventsTable().PNids; /* event found */
  else return 0; /* event not found */
  List<Expression> *expr, *dumexpr = new List<Expression>;
  for( ListItr<int> Itr(*ids); +Itr; ++Itr )  {
    PtrPN.Zeroth();
    if (PtrPN.Find(PNnode(Itr(), 1, 0,"Filname","Action",this_places,0,expr))) {
      places = PtrPN().places;
      ListItr<Place>    PtrPlace(*places);
      FilterMsg      *state = new FilterMsg;
      if (PtrPlace.Find(Place(0,0,-1,PrimEvent->id,state))) {
        if (PtrPlace().mark ==1 && PtrPlace().mode==0) { /*duplicate occurance */
          continue;
        }
        PtrPlace().mark=1; PtrPN().fire_flag--;
        if (PtrPN().fire_flag == 0)  {/* all places marked */
          if (FXEvaluation(&(PtrPN()))) { /* perform action */
            RestoreNode(&(PtrPN())); /*restore filter for reactivation*/
          }
        }
        else  continue;
      }
      else  break; /* PrimEvent not found */
    }
    else continue;
  }
  return result;
} /* end of PNEventMatch() */
```

APPENDIX D

SCALABILITY TEST SIMULATION PROGRAM

```
main()
{
  float cen,dec,hier,P,pr=0.5,LMA,DMA=0,N,Mu=8000,L,freq,D,Result;
  int i,SATURATED=0;
  for (;;)  {
    cout << "Enter N>";    cin >> N;
    cout << "Enter Event Frequency>";    cin >> freq;
    P = ((freq*N)/Mu); cen = (1/Mu) / (1-P);
    cout << "Centralized= " << cen << endl;
    dec = ((1/Mu) / (1-(freq/Mu))) + ((1/Mu) / (1-((freq*pr*N)/Mu)));
    cout << "Decentralized= " << dec << endl;
    cout << "Enter # LMAs>";  cin >> D;/*D:bracnhing factor,L:hierachy hight*/
    LMA = ((1/Mu) / (1-((pr*freq)/Mu)));
    L=log10(N)/log10(D); /* <==> L=logD(N); */
    L=ceil(log10(N));
    cout << "L=" <<L<<endl;
    for (i=1; i <= L-1; i++)    {
      Result = ((1/Mu) / (1-((freq*pr*D)/Mu)));
      DMA =  DMA + Result;
      D=D*D;
      if (Result < 0)  {
        SATURATED=1;
        break;
      }
    }
```

```
    if (SATURATED)    {
      cout << "DMAs are SATURATED ......" << endl;
    }
   else   {
     cout << "Hierarchical(0.1)= " << LMA+ (.1*DMA) << endl;
     cout << "Hierarchical(0.5)= " << LMA+(.5*DMA) << endl;
     cout << "Hierarchical(0.9)= " << LMA+(.9*DMA) << endl;
   }
    DMA=0;SATURATED=0;
  }
}
```

APPENDIX E

ACRONYMS

BSD	Berkeley Software Division
CCITT	The International Telegraph and Telephone Consultative Committee
CMIP	Common Management Information Protocols
DMA	Domain Monitoring Agent
ERS	Event Reporting Stub
ESL	Environment Specification Language
EX	Event Expression
FX	Filter Expression
HASL	High-Level Action Specification Language
HESL	High-Level Event Specification Language
HFSL	High-Level Filter Specification Language
HSN	High Speed Network
IP	Internet Protocol
IRI	Interactive Remote Instruction
LAN	Local Area Network
LMA	Local Monitoring Agent

(Continued in the next page)

LSD	Large-scale Distributed Systems
MA	Monitoring Agent
MAN	Metropolitan Area Network
MLP	Monitoring Language Processor
MSL	Monitoring System Language
SNTP	Simple Network Time Protocol
OSI	Open System Interconnection
SNMP	Simple Network Management Protocol
RFC	Request For Comments
RMP	Reliable Multicast Protocol
RSVP	Resource Reservation Protocol
RTP	Real-time Transport Protocol
RTT	Round Trip Time
TCP	Transmission Control Protocol

VITA

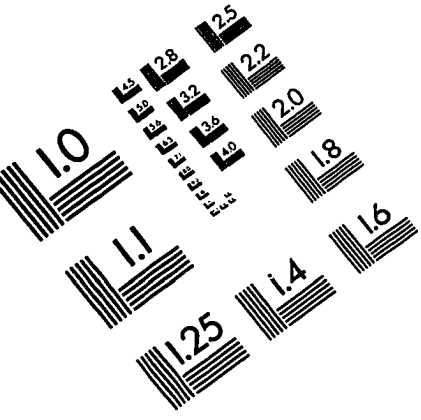
Ehab Salem Al-Shaer was born in Riyadh, Saudi Arabia, on July 4, 1967. He received his Bachelor of Science in Computer Engineering form College of Computer Science and Engineering, King Fahad University of Petroleum and Minerals, Saudi Arabia, in May 1990. Al-Shaer worked as a senior networking engineer in DataGeneral Corp. from June 1990 to July 1992. During his work in the industry, he was awarded 14 professional certificates in networking and image archiving technology from DataGeneral, Tellabs, Novell, Cygnet and Racal Melgo. In June 1992, Al-Shaer joined the computer science graduate school at Northeastern University, Boston, Massachusetts, from which he received his Master of Science degree in December 1993. Then he worked as a Research Assistant in Computer and Communication Research Center (CCRC) in Washington University until August 1995. Al-Shaer started working on his Ph.D. degree in Computer Science at Old Dominion University, Norfolk, Virginia in September, 1995. Al-Shaer was awarded fellowship grants from USENIX in 1992 and from NASA Langley Research Center in 1997. During his academic career, Al-Shaer published more than more than 10 refereed journal and conference publications in the area of network protocols, network and system management, distributed computing and object-oriented technology. Al-Shaer is a member in IEEE, ACM, ISCA, USENIX and Phi Kappa Phi Honor Society.

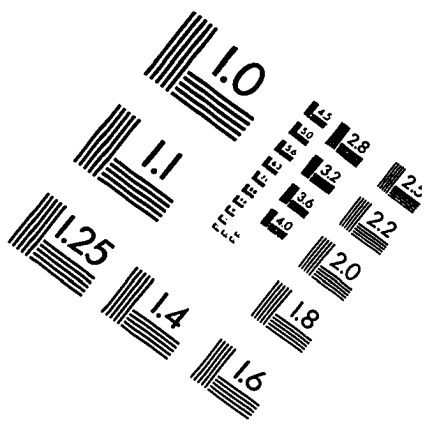
Permanent address: Department of Computer Science
Old Dominion University
Norfolk, VA 23529
USA

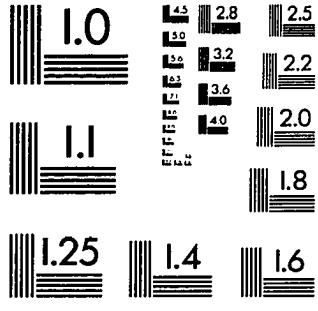
This dissertation was typeset using LaTeX* by the author.

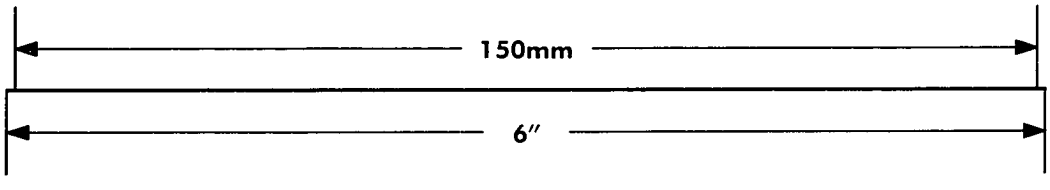
*LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.

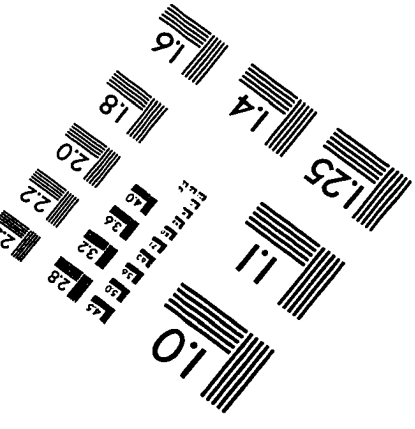
IMAGE EVALUATION TEST TARGET (QA-3)

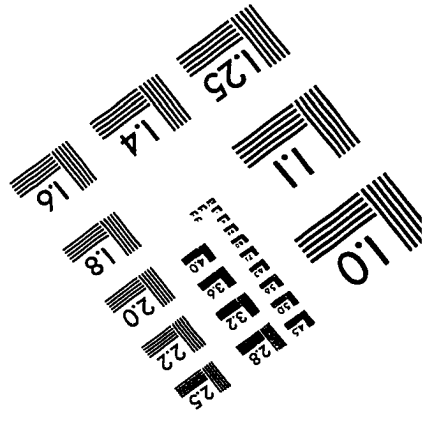












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989